图灵原创





MariaDB 原理与实现

张金鹏 张成远 季锡强 编著


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明



张金鹏

京东资深技术专家，MySQL数据库专家，京东云数据库组核心成员，主要负责MySQL内核优化及二次开发。多年数据库领域以及搜索引擎领域的工作经验，对Redis、Memcached等NoSQL数据库，以及CGroup、LXC、Docker等容器技术有深入的研究。

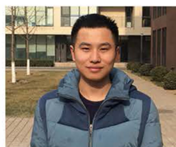
新浪微博：弓长金鹏



张成远

京东资深架构师。毕业于东北大学，硕士阶段研究分布式数据库相关方向，2012年加入京东云数据库技术团队。擅长高性能服务器开发、分布式数据库、分布式存储/缓存等大规模分布式系统架构。主导了京东分布式数据库系统及容器系统的架构与开发工作。

新浪微博：NEU_寒水



季锡强

京东资深架构师，主要负责京东分布式数据库的架构设计及研发工作，主导容器技术在京东的应用与推广。专注于高性能服务器、分布式数据库、分布式存储/缓存及容器技术，对容器核心技术及golang并发编程有深入研究。

TURING 图灵原创



MariaDB原理与实现

张金鹏 张成远 季锡强 编著

人民邮电出版社
北 京

图书在版编目（CIP）数据

MariaDB原理与实现 / 张金鹏，张成远，季锡强编著 .
-- 北京 : 人民邮电出版社，2015.3
（图灵原创）
ISBN 978-7-115-38517-8

Ⅰ. ①M… Ⅱ. ①张… ②张… ③季… Ⅲ. ①关系数
据库系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2015)第025436号

内 容 提 要

本书由浅入深地剖析了 MariaDB，首先简要介绍了一些基础知识、新特性、对 MySQL 原有功能所做的扩展以及源代码，接着介绍了底层数据结构、线程池技术、binlog、复制等内容，最后介绍了分布式数据库和京东的分布式数据库系统。

本书主要面向想了解 MariaDB/MySQL 的工作原理及具体实现的读者，以及想要阅读 MariaDB/MySQL 源代码却苦于不知道从何处开始的读者。

◆ 编　　著　张金鹏　张成远　季锡强
　责任编辑　王军花
　责任印制　杨林杰
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　印刷
◆ 开本：800×1000　1/16
　印张：18.75
　字数：443千字　　　　2015年 3 月第 1 版
　印数：1 – 4 000册　　2015年 3 月北京第 1 次印刷

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

序

自从 60 多年前关系型数据库出现起，它就被广泛使用。至今，几乎所有 IT 系统都离不开关系型数据库。在 MySQL 出现前，关系型数据库行业几乎被 Oracle、IBM DB2 和 Microsoft SQL Server 几个商业巨头垄断了。商业数据库的封闭性和高昂的价格，对 IT 行业在深度和广度上的使用、提升数据库系统以及时适应各种新需求（尤其是互联网等行业带来的新需求）造成了很大的阻碍。20 世纪末开源数据库 MySQL 的出现和崛起，为整个数据库行业带来了巨大的希望。可惜随着 MySQL 被 Oracle 公司收购，存在 Oracle 公司将 MySQL 闭源的巨大潜在风险。为了避免这种风险，为行业提供一个永久开源免费的关系型数据库系统，MariaDB 被创造了出来。虽然 MariaDB 最初只是 MySQL 的一个分支，但近年来随着社区的壮大和普及度的迅猛上升，它在很多方面已经超过了 MySQL，例如性能、复制功能和存储引擎等，并且大有完全替代 MySQL 的势头。

本书深入浅出地阐述了 MariaDB 的设计理念，剖析了 MariaDB 的几个关键而有趣的子系统。本书的作者是我的同事，我目睹了他们将日常工作实践中总结和提炼出 MariaDB 的心得转化成书的过程。这是一本理论结合实践的书，读者不但可以学习到 MariaDB 的理论，还可以参照书中的实例一步步地自行调试实践。我相信本书会对有志学习 MariaDB 的读者有所帮助。

作为一个曾经在数据库行业耕耘 8 年的老兵（主要从事 SQL Server 存储引擎研发），我也深深地期待国人不但能深度掌握 MariaDB 的使用技术，还能在 MariaDB 社区作出越来越多的贡献，为积累和提升数据库方面的核心能力而努力。期待这本书能启发和引导更多的同行来参与 MariaDB 社区的工作并发挥出价值。

何刚，京东集团技术副总裁

前　　言

目标读者

这是一本讲解数据库原理及其具体实现的书，本书主要面向想要了解 MariaDB 和 MySQL 的工作原理及其具体实现的读者，以及想要阅读 MariaDB 和 MySQL 源代码却苦于不知道从何处开始的读者。

本书的整体思路是由简至繁，从基本原理到具体实现细节，书中由浅入深地剖析了 MariaDB 以及 MySQL。

如果你想要阅读本书，首先必须具有一定的数据库基础知识；其次，你应该具备一定的 C/C++ 语言知识，因为 MariaDB/MySQL 主要是用 C/C++编写的，在分析具体实现的时候，我们会给出大量的源代码。

章节安排

本书的章节安排如下。

第 1 章描述了 MariaDB 相关的一些基础内容，包括 MariaDB 的历史、MariaDB 所做的一些优化、MariaDB 与 MySQL 的兼容性以及如何安装使用 MariaDB，等等。

第 2 章讲解了 MariaDB 添加的新特性以及对 MySQL 原有功能所做的扩展。

第 3 章首先讲解了 MariaDB 源代码的组织结构以及 MariaDB 对函数和基本类型的封装，最后从实战的角度讲解了如何调试 MariaDB。读者可以通过边调试边阅读 MariaDB 代码的方式来学习 MariaDB。

第 4 章对 MariaDB/MySQL 中使用比较多的一些底层数据结构进行了介绍，为我们后面更深入地讲解 MariaDB/MySQL 各种功能的实现奠定了基础。

第 5 章详细分析了 MariaDB 的线程池技术，包括使用以及具体的实现机制。

第 6 章从实际的应用场景出发，详细阐述了 binlog 的相关内容。

第 7 章重点讲解了 MariaDB 的 binlog group commit 技术，该技术在具有高并发事务的场景下能够极大地提高 MariaDB 单位时间内的事务提交数。

第 8 章用很大的篇幅介绍了 MariaDB/MySQL 复制相关的内容，主要包括复制的工作原理、复制的实现、半同步复制、MariaDB 多源复制、GTID，等等。

第 9 章讲解了数据库技术中用到的一些数据结构和算法，包括 B+树数据结构、`ORDER BY` 语句所使用的算法、`JOIN` 相关的算法，等等。

第 10 章介绍了分布式数据库相关的一些内容，包括基本概念、特点、一些技术难点，等等。最后，我们分析了京东分布式数据库系统的架构。

附录 A 讲解了资源控制方面的一些内容，主要是关于如何利用 Linux 系统的 CGroup 机制来控制系统资源的使用情况，并且详细分析了 CGroup 各个子系统的具体实现。

我们建议读者按顺序来阅读本书，但你也可以挑选自己感兴趣的章节进行阅读。

致谢

这里我们要感谢京东云平台开放云事业部负责人郭理靖在本书编写过程中的大力支持以及相关的指导工作，感谢京东云平台云数据库组的田琪、闫国旗、张帅、杨轩嘉、都海峰在本书的编写过程中提供的大量支持和帮助，让本书的编写工作能顺利完成。

联系作者

在编写本书的过程中，我们尽全力做到最好，仍无法避免一些错误，如果你在本书中发现了错误，可以通过新浪微博（@弓长金鹏）与我们交流，我们将非常感谢。

目　　录

第 1 章　MariaDB 概述……1

1.1　MariaDB 的历史……1

1.2　MariaDB 所做的事情……2

1.2.1　更丰富的存储引擎……2

1.2.2　性能的提升……2

1.2.3　扩展和新特性……3

1.2.4　更好的测试……3

1.2.5　尽量消除错误和警告……4

1.3　MariaDB 的版本与兼容性……4

1.3.1　MariaDB 5.1 和 MySQL 5.1 的不兼容性……4

1.3.2　MariaDB 5.2 和 MySQL 5.1 的不兼容性……5

1.3.3　MariaDB 5.3 和 MySQL 5.1、MariaDB 5.2 的不兼容性……5

1.3.4　MariaDB 5.5 和 MariaDB 5.3 的不兼容性……6

1.3.5　MariaDB 5.5 与 MariaDB 5.3 和 MySQL 5.5 的不兼容性……7

1.3.6　MariaDB 10.0 和 MySQL 5.6 的不兼容性……7

1.4　编译和安装 MariaDB……8

1.4.1　使用二进制安装包进行安装……8

1.4.2　使用源代码进行编译安装……9

1.5　联系社区……11

1.6　小结……12

第 2 章　MariaDB 的扩展和新特性……13

2.1　更多的存储引擎……13

2.1.1　全新的 Aria 存储引擎……13

2.1.2　XtraDB 存储引擎……16

2.1.3　SphinxSE 存储引擎……17

2.1.4　FederatedX 存储引擎……17

2.1.5　TokuDB 存储引擎……18

2.1.6　Cassandra 存储引擎……18

2.1.7　CONNECT 存储引擎……18

2.1.8　Sequence 存储引擎……19

2.1.9　Spider 存储引擎……20

2.2　线程池技术和 binlog group commit 技术……22

2.2.1　线程池技术……22

2.2.2　binlog group commit 技术……23

2.3　MariaDB 其他扩展和新特性……23

2.3.1　更高的时间精度……23

2.3.2　虚拟列……24

2.3.3　用户统计功能……25

2.3.4　KILL 命令的扩展……27

2.3.5　命令执行进度报告……27

2.3.6　动态列……28

2.3.7　多源复制……29

2.4　小结……29

第 3 章　初识 MariaDB 源代码……31

3.1　MariaDB 源代码的目录组织结构……31

3.2　MariaDB 对类型和函数的封装……33

3.2.1　对类型的封装……33

3.2.2　对函数的封装……33

3.3　调试 MariaDB……34

3.3.1　准备工作……34

3.3.2　mysqld 关键的函数调用……37

3.3.3　调试……38

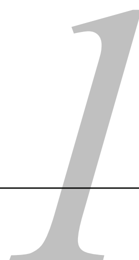
3.4　小结……40

第 4 章 MariaDB 基础数据结构 41
4.1 内存池 MEM_ROOT 41
4.1.1 内存碎片问题 42
4.1.2 MEM_ROOT 的定义 44
4.1.3 MEM_ROOT 的使用 46
4.1.4 MEM_ROOT 的初始化 47
4.1.5 分配内存 48
4.1.6 内存回收 50
4.1.7 MEM_ROOT 的使用场景 52
4.2 文件缓存 IO_CACHE 52
4.2.1 高性能武器——缓存 53
4.2.2 IO_CACHE 的定义 54
4.2.3 IO_CACHE 的使用 57
4.3 NET 结构 63
4.4 线程上下文——THD 65
4.5 TABLE_SHARE 69
4.6 TABLE 73
4.7 小结 76
第 5 章 MariaDB 线程池 77
5.1 线程池相关的参数 77
5.1.1 MariaDB 5.1 和 MariaDB 5.3 中的线程池 77
5.1.2 MariaDB 5.5 和 MariaDB 10.0 中的线程池 78
5.2 何时使用线程池 79
5.3 线程池的实现 79
5.3.1 线程池相关的数据结构 80
5.3.2 线程池的初始化 82
5.3.3 添加连接到线程池 84
5.3.4 worker 线程 85
5.3.5 get_event 函数 86
5.3.6 listener 线程 89
5.3.7 timer 线程 92
5.4 小结 94
第 6 章 二进制日志 binlog 95
6.1 简介 95
6.1.1 binlog 的作用 96
6.1.2 index 文件 96
6.2 binlog 的使用 97
6.2.1 开启 binlog 97
6.2.2 选择 binlog 的格式 97
6.2.3 binlog 的相关参数 98
6.3 binlog 事件 99
6.3.1 binlog 事件格式 99
6.3.2 binlog 事件类型 101
6.3.3 binlog 事件的实现 108
6.4 清理 binlog 109
6.4.1 手动清理 binlog 109
6.4.2 自动清理 binlog 109
6.4.3 purge 命令的实现 109
6.5 binlog_cache_mngr 结构 112
6.6 mysqlbinlog 工具 112
6.7 使用 binlog 进行恢复 113
6.8 小结 113
第 7 章 binlog group commit 技术 115
7.1 事务的两阶段提交 115
7.2 binlog group commit 的工作原理 117
7.3 binlog group commit 的实现 118
7.3.1 相关的数据结构 118
7.3.2 代码执行流程 120
7.3.3 事务排队 125
7.3.4 leader 线程的工作 128
7.3.5 prepare_ordered 和 commit_ordered 接口 134
7.4 小结 135
第 8 章 复制 136
8.1 简介 136
8.2 复制的作用 137
8.3 复制的工作原理 138
8.3.1 概要 138
8.3.2 relay-log 140
8.3.3 master.info 文件和 relay-log.info 文件 140
8.4 复制的配置 141
8.4.1 在新安装的主库和从库上配置复制 141

8.4.2 主库有一定数据时的复制配置……144
8.4.3 选择性复制……150
8.5 复制的实现……151
8.5.1 复制相关的数据结构……152
8.5.2 复制的初始化——init_slave函数……157
8.5.3 CHANGE MASTER TO命令——准备工作……159
8.5.4 START SLAVE命令——开启复制……160
8.5.5 STOP SLAVE命令——停止复制……160
8.5.6 slave IO线程……161
8.5.7 slave SQL线程……164
8.5.8 master dump线程……165
8.6 半同步复制……168
8.6.1 半同步复制的工作原理……168
8.6.2 半同步的安装和配置……169
8.6.3 半同步复制的实现……171
8.6.4 半同步复制的变种……179
8.6.5 半同步复制的潜在问题……180
8.7 并行复制……181
8.7.1 MySQL的并行复制……181
8.7.2 MariaDB的并行复制……181
8.8 多源复制……182
8.8.1 多源复制的应用场景……182
8.8.2 多源复制相关的命令……183
8.8.3 MariaDB多源复制的实现……184
8.9 GTID……185
8.9.1 GTID的概念……186
8.9.2 在MySQL上配置基于GTID的复制……186
8.9.3 MySQL中GTID的实现……187
8.9.4 MariaDB中的GTID……195
8.10 小结……195
第9章 数据结构和算法……197
9.1 算法复杂度……197
9.2 B+树和索引……198
9.2.1 磁盘的读取……198
9.2.2 B+树……199
9.2.3 数据库索引……200
9.3 堆排序与快速排序……201
9.3.1 堆——优先级队列……201
9.3.2 堆排序……202
9.3.3 快速排序——qsort……203
9.4 ORDER BY的实现……204
9.4.1 使用索引的已有顺序……205
9.4.2 filesort算法……207
9.5 JOIN的实现……210
9.5.1 JOIN语句的使用……211
9.5.2 Nest Loop Join算法……212
9.5.3 Block Nest Loop Join算法……214
9.5.4 Batched Key Access Join算法……216
9.5.5 Hash Join算法……216
9.5.6 Sort Merge Join算法……217
9.6 小结……218
第10章 分布式数据库……219
10.1 分布式数据库概要……219
10.1.1 分布式数据库的特点……219
10.1.2 系统的扩展方式……220
10.1.3 分布式数据库中的技术难点……221
10.2 数据的分片方式……221
10.3 分布式数据库实践——京东分布式数据库系统……222
10.3.1 京东分布式数据库系统架构……222
10.3.2 高可用组的初始化……223
10.3.3 数据的分片……224
10.3.4 系统的高可用性……225
10.3.5 系统的可扩展性……227
10.4 小结……230
附录A 数据库与IO资源控制……231

第 1 章

MariaDB概述



2008年MySQL首先被Sun公司收购，之后Sun公司又被Oracle公司收购，MySQL也被包含在这次收购中。在这两次收购过程中，出现了多个MySQL的开源分支，其中比较主流的分支有Percona Server、MariaDB和Drizzle等。它们都有活跃的用户社区和一定程度上的商业支持，均由独立的服务供应商支持。

MariaDB是众多MySQL开源分支中非常出色的一个。作为MySQL的深度替代者，MariaDB很好地兼容了MySQL。同时，MariaDB在MySQL的基础上做了很多扩展，包含了许多新特性，例如支持binlog group commit技术，支持虚拟列和动态列，支持多源复制，等等。在性能方面，MariaDB也做了很多优化，例如更快的子查询、更快的字符集转换、为MyISAM存储引擎添加了分段键值缓存，让MyISAM存储引擎在现代硬件体系上运行得更快，等等。本章中，我们将介绍MariaDB的一些基本信息。

本章的内容主要包括：

- MariaDB的历史
- MariaDB所做的事情
- MariaDB的版本与兼容性
- 编译和安装MariaDB
- 联系社区

1.1　MariaDB 的历史

在Sun收购MySQL之后，MySQL的创始人Monty Widenius离开了Sun公司，成立了Monty程序公司，创立了MariaDB，其主要目的是建立一个开放的开发环境，以鼓励外部人员参与。目前，MariaDB主要由社区开发和维护。

MariaDB对社区、开发者及用户的主要意义可以概括为以下几个方面：

- 是一个永久开源的MySQL分支；

- 高质量和持续性的开发测试及维护工作；
- 来自社区开发者提交的补丁会被确认、接收和使用；
- 维护MariaDB整个社区开发者的话语权，不会被某一个人或者组织完全控制；
- 持续保持和MySQL的兼容性。

1.2　MariaDB 所做的事情

MariaDB在MySQL的基础上做了许多工作，主要包括性能方面的优化以及许多新特性的支持。

1.2.1　更丰富的存储引擎

除了包含标准的MyISAM、BLACKHOLE、CSV、MEMORY、ARCHIVE、MERGE等存储引擎外，MariaDB的源代码和二进制包中还包含以下额外的存储引擎：

- Aria
- XtraDB
- PBXT
- FederatedX
- OQGraph
- SphinxSE
- IBMDB2I
- TokuDB
- Cassandra
- CONNECT
- Sequence
- Spider

1.2.2　性能的提升

MariaDB在很多地方都做了优化，其性能得到了不同程度的提升。MariaDB所做的优化主要包括以下几个方面。

- 对子查询进行了优化，使子查询的速度得到了提升。
- 更快、更安全的复制，引入了binlog group commit技术，使MariaDB在某些场景下事务的提交速度得到成倍提升。
- 提升了在Windows平台下InnoDB异步IO子系统的性能。
- 提高了MEMORY存储引擎索引的插入速度。

- 为MyISAM存储引擎增加了分段键值缓存，提升了MyISAM存储引擎在现代硬件上的运行速度。
- 表的校验更加快速。
- 提升了字符集的转换效率。
- 引入了线程池技术，解决了MySQL的最大连接数限制问题，降低了大量连接情况下的系统开销。
- 使用Aria存储引擎的表作为临时表，提升了某些复杂查询的效率。

1.2.3 扩展和新特性

MariaDB是由社区开发和维护的，主要由用户的需求所驱动，如果某个功能对用户是有用的，MariaDB就会考虑将其加入进来，所以MariaDB包含了许多扩展和新特性。目前，MariaDB比较显著的扩展和特性主要包括以下几个。

- 时间精度达到微秒级别。
- 虚拟列的支持。
- 动态列的支持。
- 用户统计的扩展。
- `KILL`命令的扩展，允许杀死某个用户的所有查询。
- `CREATE TABLE`命令的扩展。
- INFORMATION_SCHEMA.PLUGINS表的增强。
- 引入了binlog group commit技术。
- 命令的执行进度报告。
- 更快的连接和子查询。
- GIS功能的支持。
- 支持多源复制，允许一个从库同时复制多个主库的数据。
- GTID的支持，简化了复制的运行和维护工作。
- 支持PCRE正则表达式。

1.2.4 更好的测试

MariaDB在测试方面也做了很多工作，主要表现在以下几个方面。

- 更多的测试。
- bug修复的测试。
- 配置不同的编译选项，获得更好的测试效果。
- 移除无效的测试。

1.2.5　尽量消除错误和警告

为了避免潜在的问题，在编译程序的时候，不要忽略编译器的警告信息，MariaDB在这方面做得非常好，具体如下所示。

- 修复尽可能多的错误，避免引入新的错误。
- 编译器的警告是不好的，消除尽可能多的警告。

1.3　MariaDB 的版本与兼容性

出于实用的目的，MariaDB是相同版本MySQL的二进制深度替代者，例如MariaDB 5.1、MariaDB 5.2和MariaDB 5.3对应于MySQL 5.1，MariaDB 5.5 对应于MySQL 5.5，MariaDB 10.0对应于MySQL 5.6。

MariaDB与MySQL的兼容性主要体现在以下几个方面。

- 数据文件和表定义文件是二进制兼容的。
- 所有的客户端API和协议都是兼容的。
- 所有的文件名、二进制文件、路径、端口号等都是相同的。
- 所有的连接器，包括PHP、Perl、Python、Java、.NET、Ruby、MySQL的连接器在MariaDB上都是可以正常使用的，不需要进行任何改动。
- 可以使用MySQL的客户端连接到MariaDB上。

也就是说，在大多数情况下，卸载已有的MySQL并安装对应版本的MariaDB，就可以工作得很好，不需要转换任何的数据文件，前提是使用对应版本的MariaDB，例如使用MariaDB 5.1替换MySQL5.1。同时，你必须还要运行mysql_upgrade来完成升级，确保你的MySQL权限和事件表更新了MariaDB的新字段。

MariaDB每个月都会与MySQL代码库合并来确保兼容性，并添加Oracle修正的bug和特性。MariaDB在脚本升级方面也做了大量的工作，从MySQL 5.0升级到MariaDB 5.1比从MySQL 5.0升级到MySQL 5.1更容易。

1.3.1　MariaDB 5.1 和MySQL 5.1 的不兼容性

为了使MariaDB提供更多、更有用的信息，在极少的一些情况下会导致MariaDB和MySQL不兼容。下面列出了从用户角度来看MariaDB 5.1与MySQL 5.1的所有不兼容性。

- 安装包的名称以MariaDB开头，而不是以MySQL开头。
- 在my.cnf配置文件中，可以使用`[mariadb]`来替代`[mysqld]`。

- 在MariaDB中加载其他二进制的存储引擎时，如果该存储引擎不是使用对应的MariaDB版本编译的，那么该二进制的存储引擎将不可用。这是因为服务器的内部数据结构THD在MariaDB和MySQL中是不同的，而且在MariaDB/MySQL的不同版本中也是不同的。通常这不是问题，因为对于大多数人来说，不需要加载二进制的存储引擎，MariaDB本身就拥有比MySQL更多的存储引擎。
- 表的校验可能产生不同的结果，因为MariaDB在校验的时候并不忽略NULL列（新式校验方法），而MySQL 5.1在校验的时候会忽略NULL列（旧式校验方法）。在MariaDB中，开启`mysqld --old`选项，你将会得到旧式的校验结果。但需要注意的是，MyISAM存储引擎和Aria存储引擎内部使用的是新式校验方式，所以当你使用了`--old`选项时，`CHECKSUM`命令将会执行得很慢，因为执行该命令的时候需要一行一行地扫描表的数据，然后按照“旧式”的方法生成校验结果。
- MariaDB的慢查询日志包含了更多关于查询的信息，如果你使用已有的慢查询日志解析脚本对MariaDB的慢查询日志进行解析可能会出现问题。
- MariaDB占用的内存通常会比MySQL多一点，因为在默认情况下，MariaDB会启用Aria存储引擎来操作内部临时表。如果想让MariaDB占用较少的内存（这将会牺牲一些性能），你可以设置`aria_pagecache_buffer_size`的值为1MB（默认值为128MB）。
- 如果你正在使用MariaDB的新选项、新特性或者新存储引擎，那么就不能在MySQL和MariaDB之间进行切换。

1.3.2 MariaDB 5.2 和MySQL 5.1 的不兼容性

除了上一节中列出的MariaDB 5.1和MySQL 5.1的不兼容性之外，MariaDB 5.2还有一些地方与MySQL 5.1不兼容。例如新增`SQL_MODE`的取值`IGNORE_BAD_TABLE_OPTIONS`，该选项会忽略由于存储引擎不支持某些选项而导致的错误。

1.3.3 MariaDB 5.3 和MySQL 5.1、MariaDB 5.2 的不兼容性

MariaDB 5.3不仅与MySQL 5.1在某些地方不兼容，同时与MariaDB 5.2也有一些地方不兼容。

- 由于转换而导致的错误，MariaDB提供了更加详细的说明。
- MariaDB的错误编号已经从1900开始，目的是避免与MySQL的错误编号产生冲突。
- MariaDB从5.3开始，内部使用的时间精度为微秒，而在MySQL内部以及5.3以前版本的MariaDB中，时间精度为毫秒。
- 在MariaDB中返回的是包含6位小数的时间戳，但是MySQL返回的时间戳是不带小数的。当你使用`UNIX_TIMESTAMP` 作为分区函数时，会导致一些问题。修复这些问题可以使用`FLOOR(UNIX_TIMESTAMP())`函数代替或者是将日期字符串改成日期数字，如20080101000000。
- MariaDB对于类型`date`、`datetime`和`timestamp`值的检查更加严格，如`UNIX_TIMESTAMP("x")`返回的是`NULL`，而不是0。

- SHOW PROCESSLIST 拥有一个额外的进度列，显示了命令的执行进度。通过启动mysqld时携带--old-mode=NO_PROGRESS_INFO或者--old选项来禁用该功能。
- INFORMATION_SCHEMA.PROCESSLIST表新增了3个字段用于扩展命令执行进度报告功能，即STAGE、MAX_STAGE和PROGRESS。
- 若长注释以/*M!或者/*M! #####开头，注释内的命令将会被执行。
- MariaDB在启动mysqld时，如果使用了max_user_connections=0（即不对连接数加以限制）参数，那么在mysqld运行的时候不能改变全局变量max_user_connections的值。这是因为启动mysqld时带上max_user_connections=0，MariaDB内部不会分配计数结构。如果之后改变了这个变量，将会导致错误的计数。
- 可以将max_user_connections设置为–1阻止用户连接服务器，但拥有SUPER权限的用户还是可以连接上的。
- IGNORE指令不会忽略所有的错误，仅仅会忽略安全的错误。

1.3.4　MariaDB 5.5 和MariaDB 5.3 的不兼容性

XtraDB存储引擎之前版本中的某些选项在XtraDB 5.5中将不再支持，具体包括以下几个方面。

- innodb_adaptive_checkpoint：使用innodb_adaptive_flushing_method替代。
- innodb_auto_lru_dump：使用innodb_buffer_pool_restore_at_startup替代。
- innodb_blocking_lru_restore：使用innodb_blocking_buffer_pool_restore替代。
- innodb_enable_unsafe_group_commit
- innodb_expand_import：使用innodb_import_table_from_xtrabackup替代。
- innodb_extra_rsegments：使用innodb_rollback_segment替代。
- innodb_extra_undoslots
- innodb_fast_recovery
- innodb_flush_log_at_trx_commit_session
- innodb_overwrite_relay_log_info
- innodb_pass_corrupt_table：使用innodb_corrupt_table_action替代。
- innodb_use_purge_thread
- xtradb_enhancements

XtraDB 5.5的某些选项的默认值发生了改变，主要有以下几项。

- innodb_change_buffering的旧默认值为inserts，新默认值为all。
- innodb_flush_neighbor_pages的旧默认值为1，新默认值为area。

XtraDB 5.5添加了一些新的选项，具体如下：

- innodb_adaptive_flushing_method

- innodb_adaptive_hash_index_partitions
- innodb_blocking_buffer_pool_restore
- innodb_buffer_pool_instances
- innodb_buffer_pool_restore_at_startup
- innodb_change_buffering_debug
- innodb_corrupt_table_action
- innodb_flush_checkpoint_debug
- innodb_force_load_corrupted
- innodb_import_table_from_xtrabackup
- innodb_large_prefix
- innodb_purge_batch_size
- innodb_purge_threads
- innodb_recovery_update_relay_log
- innodb_rollback_segments
- innodb_sys_columns
- innodb_sys_fields
- innodb_sys_foreign
- innodb_sys_foreign_cols
- innodb_sys_tablestats
- innodb_use_global_flush_log_at_trx_commit
- innodb_use_native_aio

1.3.5 MariaDB 5.5 与MariaDB 5.3 和MySQL 5.5 的不兼容性

除了1.3.4节介绍的XtraDB在MariaDB 5.5和MariaDB 5.3中的不兼容性，MariaDB 5.5与MariaDB 5.3和MySQL 5.5还具有以下几个不兼容的地方。

- INSERT IGNORE会对重复键值给出警告信息。通过设置OLD_MODE=NO_DUP_KEY_WARNINGS_WITH_IGNORE，可以关闭INSERT IGNORE对重复键值的警告。
- X'HHHH'在标准SQL语法中是用来表示二进制字符串的，但在MariaDB 5.5.31之前，它将被错误地理解为数字，和0xHHHH的作用一致。而在MariaDB 5.5.31中，该问题已被修复了，X'HHHH'只能表示字符串。

1.3.6 MariaDB 10.0 和MySQL 5.6 的不兼容性

作为MariaDB目前最新的版本，MariaDB 10.0与MySQL 5.6的不兼容性包括以下几个方面。

- 如果仅仅给出了选项的前缀部分，例如使用--big-table取代--big-tables，MySQL将会给出警告，而MariaDB将会正常工作。也就是说，只要给出的前缀部分能够唯一地标识该选项即可。
- MariaDB的GTID和MySQL 5.6的GTID不能兼容，也就是说MySQL 5.6不能作为MariaDB 10.0的从库。
- 为了使CREATE TABLE ... SELECT命令在基于行模式复制和基于命令模式复制的情况下都能正常工作，MariaDB中CREATE TABLE ... SELECT命令在从库上将会被转化成CREATE OR RPLACE命令执行。这样的好处是即便从库在执行CREATE TABLE ... SELECT命令时宕机了，仍然能够正常工作。

1.4　编译和安装 MariaDB

在使用MariaDB或者学习MariaDB之前，建议首先在你的机器上安装MariaDB。安装方式主要有两种，一是使用二进制安装包进行安装，二是使用源代码进行编译安装，下面我们分别进行介绍。

1.4.1　使用二进制安装包进行安装

使用二进制安装包安装MariaDB是非常简单的，主要步骤如下。

(1) 根据自己的系统平台，前往MariaDB官方网站获取MariaDB对应的二进制安装包：https://downloads.mariadb.org/。

(2) 解压下载的安装包，将解压后的二进制文件复制到指定的目录：

```
shell> tar zxvf mariadb-10.0.6-x86_64.tar.gz
shell> cp -r mariadb-10.0.6-x86_64/* /usr/local/mariadb10/
```

(3) 创建MariaDB用户组和用户：

```
shell> groupadd maria
shell> useradd -r -g maria maria
```

(4) 改变目录权限：

```
shell> cd /usr/local/mysql
shell> sudo chown -R maria .
shell> sudo chgrp -R maria .
```

(5) 初始化MariaDB系统表：

```
shell> sudo scripts/mysql_install_db --user=maria
shell> sudo chown -R maria data
```

(6) 启动MariaDB server进程：

```
shell> cd /usr/local/mysql
shell> sudo bin/mysqld_safe --user=maria &
```

1.4.2 使用源代码进行编译安装

使用源代码编译安装MariaDB主要包括以下几个步骤。

(1) 获取MariaDB源代码包。前往MariaDB官方网站的下载页面（https://downloads.mariadb.org/）下载MariaDB的源代码。

(2) 解压MariaDB源代码包：

```
shell> tar zxvf mariadb-10.0.6.tar.gz
```

(3) 编译与安装MariaDB。

MariaDB 5.5以及更高的版本使用了cmake工具进行编译，具体的步骤如下所示。

(a) 首先你可以使用cmake的默认配置，也可以为cmake添加选项：

```
shell> cd mariadb-10.0.6
shell> cmake .
```

(b) 运行完cmake之后，建立和安装：

```
shell> make
shell> sudo make install
```

(c) 创建MariaDB用户组和用户：

```
shell> groupadd maria
shell> useradd -r -g maria maria
```

(d) 改变目录权限：

```
shell> cd /usr/local/mysql
shell> sudo chown -R maria .
shell> sudo chgrp -R maria .
```

(e) 初始化MariaDB系统表：

```
shell> sudo scripts/mysql_install_db --user=maria
shell> sudo chown -R maria data
```

(f) 启动MariaDB server进程：

```
shell> cd /usr/local/mysql
shell> sudo bin/mysqld_safe --user=maria &
```

(4) 连接到MariaDB server。此时，MariaDB已经安装完成，并且处于运行状态，可以使用客户端连接并进行操作：

```
shell> cd /usr/local/mysql
shell> bin/mysql -uroot
```

在执行cmake命令生成Makefile的时候，可以根据自己的需求为cmake添加不同的参数，达到定制Makefile的目的。执行cmake . -Lh命令，可以查看cmake可选的参数。表1-1列出了cmake可以携带的所有参数，并给出了它们的含义和默认值。

表1-1　cmake参数列表

参　数	描　述	默 认 值
CMAKE_BUILD_TYPE	编译类型，可选的值包括：Debug、Release、RelWithDebInfo、MinSizeRel	RelWithDebInfo
CMAKE_INSTALL_PREFIX	安装目录前缀	/usr/local/mysql
COMMUNITY_BUILD	是否是社区共建	ON
CONNECT_WITH_LIBXML2	是否支持使用LIBXML2连接存储引擎	ON
CONNECT_WITH_MYSQL	是否支持远程MySQL连接	ON
CONNECT_WITH_ODBC	是否支持ODBC连接	ON
CONNECT_WITH_XMAP	是否支持索引文件映射方式连接	ON
ENABLE_DEBUG_SYNC	是否启用同步调试	ON
ENABLE_GCOV	是否包含GCOV支持	ON
ENABLED_PROFILING	是否启用代码查询分析	ON
INSTALL_LAYOUT	安装目录布局。选项有STANDALONE、RPM、DEB、SVR4	STANDALONE
MYSQL_DATADIR	MySQL数据目录	/usr/local/mysql/data
ODBC_INCLUDE_DIR	包含sql.h文件的目录	/usr/include
ODBC_LIBRARY	指定ODBC驱动管理库	/usr/lib/libodbc.so
TMPDIR	存放MariaDB临时文件的路径	
USE_ARIA_FOR_TMP_TABLES	使用Aria临时表	ON
WITH_ARCHIVE_STORAGE_ENGINE	归档静态连接到服务器	OFF
WITH_ARIA_STORAGE_ENGINE	是否安装Aria存储引擎	ON
WITH_BLACKHOLE_STORAGE_ENGINE	是否安装BLACKHOLE存储引擎	OFF
WITH_CONNECT_STORAGE_ENGINE	是否安装CONNECT存储引擎	OFF
WITH_EMBEDDED_SERVER	是否在嵌入式服务下编译MariaDB	OFF
WITH_EXTRA_CHARSETS	额外的字符集	all
WITH_FEDERATEDX_STORAGE_ENGINE	是否安装FederatedX存储引擎	OFF
WITH_FEEDBACK	是否安装FEEDBACK存储引擎	OFF
WITH_LIBWRAP	是否支持libwrap	OFF
WITH_LOCALES	是否静态关联LOCALES	OFF
WITH_METADATA_LOCK_INFO	是否静态关联METADATA_LOCK_INFO	OFF
WITH_PARTITION_STORAGE_ENGINE	是否安装PARTITION存储引擎	ON

（续）

参　　数	描　　述	默 认 值
WITH_PERFSCHEMA_STORAGE_ENGINE	是否安装PERFSCHEMA存储引擎	ON
WITH_QUERY_CACHE_INFO	是否静态关联QUERY_CACHE_INFO	OFF
WITH_QUERY_RESPONSE_TIME	是否静态关联QUERY_RESPONSE	OFF
WITH_READLINE	使用捆绑的readline	OFF
WITH_SEMISYNC_MASTER	是否静态关联SEMISYNC_MASTER	OFF
WITH_SEMISYNC_SLAVE	是否静态关联SEMISYNC_SLAVE	OFF
WITH_SEQUENCE_STORAGE_ENGINE	是否安装Sequence存储引擎	OFF
WITH_SPHINX_STORAGE_ENGINE	是否安装Sphinx存储引擎	OFF
WITH_TEST_SQL_DISCOVERY_STORAGE_ENGINE	是否安装TEST_SQL_DISCOVERY存储引擎	OFF
WITH_UNIT_TESTS	是否编译MariaDB单元测试	ON
WITH_VALGRIND	是否支持VALGRIND	OFF
WITH_XTRADB_STORAGE_ENGINE	是否安装XtraDB存储引擎	ON
WITH_ZLIB	是否支持zlib	system

1.5 联系社区

MariaDB是由社区进行开发和维护的，当你碰到MariaDB的相关问题时，可以与社区取得联系。下面我们给出几个社区的联系方式。

- Launchpad团队和邮件列表。MariaDB项目托管在Launchpad上，一个好的开始是去加入Launchpad团队。下面简要介绍一下各个团队。
  - maria-discuss团队和邮件列表，适合MariaDB用户和一般讨论。
  - maria-docs团队和邮件列表，适合对于文档感兴趣的人。
  - maria-developers团队和邮件列表，适合一些想贡献代码或是密切关注MariaDB开发动态的人。
  - maria-captains团队和邮件列表，适合受信任的开发人员将代码合并到官方的分支。
- MariaDB Commits列表。提交邮件发送到commit@mariadb.org。任何人都能够订阅提交邮件或是查阅相关提交档案。
- MariaDB Captains。Maria-captains是MariaDB核心开发团队。只要有足够的技术、技能并且积极参与MariaDB开发，都能成为这个团队里的一员。
- MariaDB公告列表。如果你只是想知道最新MariaDB的消息，如MariaDB的最新发行版本，你可以订阅annouce@mariadb.org，也可以查阅公告档案信息。
- IRC。Maria项目官方IRC频道是irc.freenode.net/maria。irc:irc.freenode.net/maria-meeting频道用于官方会议。为了用户的利益，官方并没有为开发者单独设立一个频道。

- 讨论组。你可以在许多的开源讨论组里找到研究MariaDB的人，如O'Reilly MySQL讨论组、OSCON和Open Source bridge。也可以参考MariaDB开发者所维护的讨论组列表：https://mariadb.org/en/ conference/。
- 要求增加新特性。如果想要新增一个特性，既可以在MariaDB官方网站的JIRA系统中提交，也可以自己完成这个特性并且成为MariaDB的开发者。
- MariaDB知识基础。MariaDB知识基础是一个手册，也是一个帮助文档，在这里你能提问和回答关于特定MariaDB/MySQL的一些问题。

1.6　小结

本章中，我们介绍了MariaDB的一些基本信息，首先概览了MariaDB的历史以及MariaDB主要做的一些事情，接着介绍了MariaDB和对应版本的MySQL的兼容性，最后讨论了如何对MariaDB进行编译和安装。

第 2 章

MariaDB的扩展和新特性

作为MySQL的深度替代者，MariaDB的主要宗旨是朝着用户和开发者所期望的方向发展，只要是用户需要的、对用户有用的特性都会被添加进来。随着MariaDB的发展，越来越多有用的特性被加入进来。例如，更多的存储引擎被添加到MariaDB中用于满足不同的用户需求，binlog group commit技术大大提高了MariaDB在高并发事务情况下的性能，通过线程池技术让MariaDB支撑更多的连接，等等。本章中，我们将介绍MariaDB所包含的一些新特性以及对MySQL原有功能做的一些扩展。

本章的内容主要包括：

- 更多的存储引擎
- 线程池技术和binlog group commit技术
- MariaDB其他扩展和新特性

2.1 更多的存储引擎

MariaDB一个很显著的特点就是除了包含一些标准存储引擎（例如MyISAM、InnoDB、Heap、Blackhole、Archive、CSV等）外，还包含了许多额外的具有不同作用的存储引擎，并且包含了自己特有的存储引擎Aria。你可以通过在MariaDB源代码的storage目录下执行ls命令来查看所有存储引擎，如图2-1所示。

```
jinpengzhang@jinpengzhang:~/mariadb-10.0.6/storage$ ls
archive    csv          heap      myisammrg   sequence             tokudb
blackhole  example      innobase  ndb         sphinx               xtradb
cassandra  federated    maria     oqgraph     spider
connect    federatedx   myisam    perfschema  test_sql_discovery
```

图2-1　MariaDB的存储引擎

2.1.1 全新的Aria存储引擎

Aria是MariaDB的一个全新的存储引擎，它是作为MyISAM存储引擎的替代者而开发的。该存储引擎从2007年开始开发，其开发者是开发MySQL server、MyISAM、MERGE以及MEMORY

存储引擎的主要成员。

目前，Aria存储引擎的最新版本是1.5，在安全方面，它拥有崩溃自动恢复功能，比MyISAM更安全；在性能方面，Aria有比MyISAM更好的缓存系统，所以相对于MyISAM有一定的提升。Aria目前还不支持事务，在未来的版本中（2.0）将完全支持事务。

1. Aria名字的由来

在Maria项目刚建立的时候，Monty（MySQL和MyISAM的创始人）和最初的开发者只是想开发下一代MyISAM存储引擎，使其具有崩溃恢复功能，并且能够提供事务的支持。Monty以他女儿的名字Maria作为该项目以及该存储引擎的名字。项目按部就班地进行着，随着项目的进行，事情发生了变化，开发者们不仅仅是工作在Maria存储引擎上，而是工作在MySQL的一个完整分支上，既然这个项目叫Maria，那么这个MySQL的分支就很自然地被命名为MariaDB，这就是MariaDB的诞生过程。后来为了区分MariaDB与Maria存储引擎，Maria存储引擎被重新命名为Aria。

2. Aria的相关参数

- 创建Aria表的可选参数

Aria存储引擎为CREATE TABLE命令提供了一些额外的选项供用户选择，具体如下所示。

- ❑ TRANSACTION= 0 | 1。在创建表时，你可以通过指定该选项的值来决定是否让你的表支持崩溃恢复。在默认情况下，该选项的值为0，也就是说默认情况下创建的Aria表是不支持崩溃恢复的。如果指定了该选项的值为1，所有的修改都会被记录到事务日志中。在这种情况下，会降低写入和更新的速度，但当系统崩溃重启时，能够根据事务日志的内容对执行到一半的命令进行回滚，使数据库恢复到命令执行前的状态。
- ❑ PAGE_CHECKSUM= 0 | 1。数据块和索引块是否添加额外的校验。
- ❑ TABLE_CHECKSUM= 0 | 1。表是否添加额外的校验。
- ❑ ROW_FORMAT= PAGE。Aria存储引擎除了支持MyISAM的所有行格式（FIEXD和DYNAMIC）外，还支持页模式的行格式，所有的数据和索引存储在页内，页模式只有在TRANSACTION=1的时候才会生效。在Aria的缓存机制中，页模式下缓存的是一个个的页。

如果你想在Aria中创建一个类似于MyISAM的非事务表,可以通过设置以下参数来达到目的:

```
CREATE TABLE t1 (a int) ROW_FORMAT=FIXED TRANSACTIONAL=0 PAGE_CHECKSUM=0;
```

或者

```
CREATE TABLE t1 (a int) ROW_FORMAT=DYNAMIC TRANSACTIONAL=0 PAGE_CHECKSUM=0;
```

- mysqld中与Aria相关的可选参数

在mysqld启动的时候，有一些与Aria存储引擎相关的选项，如表2-1所示。

表2-1 mysqld启动时与Aria存储引擎相关的选项

选　项	描　述	默 认 值
--aria[=#]	使用或者禁用Aria插件，可选的值为ON、OFF、FORCE（如果加载失败，mysqld将启动失败）	ON
--aria-block-size=#	Aria索引页的大小	8192
--aria-checkpoint-interval=#	多久创建一次检查点，单位为秒	30
--aria-force-start-after-recovery-failures=#	当根据日志执行错误恢复时，如果连续出现该参数指定个数的错误时，将采用删除日志的方法来解决问题。0代表不使用该特性	0
--aria-group-commit=#	指定Aria存储引擎的group commit（gc）的模式，可选的值有none（不使用gc）、hard（等待提交完成）、soft（不等待提交完成，存在风险）	none
--aria-group-commit-interval=#	group commit的时间间隔，单位为微妙（1/1000000秒）。该选项只有在group commit开启的状态下才能生效	0
--aria-log-dir-path= name	存储事务日志的文件夹	datadir
--aria-log-file-size=#	事务日志文件的最大容量	1073741824
--aria-log-purge-type= name	Aria事务日志的清理策略。可选的值为immediate、external和at_flush	immediate
--aria-max-sort-file-size=#	在执行ORDER BY或者GROUP BY命令时，当结果集比较大时会使用临时文件	9223372036853727232
--aria-page-checksum	页面校验	TRUE
--aria-pagecache-age-threshold=#	页缓存中，被评为热块（hot block）的阈值。页缓存中的页命中次数达到该值后才能被认定为是热块	300
--aria-pagecache-buffer-size=#	Aria表的数据库和索引块使用的缓存区大小	134217720 (= 128MB)
--aria-pagecache-division-limit=#	暖块（warm block）的最小百分比	100
--aria-recover[=#]	指定损坏的表如何自动修复，可选的值有NORMAL、BACKUP、FORCE、QUICK和OFF	NORMAL
--aria-repair-thread=#	修复表时使用的线程数	1
--aria-sort-bufer-size=#	当Aria进行表修复时执行的索引排序以及用户调用CREATE INDEX或ALTER TABLE创建索引时使用的缓存区大小	134217720
--aria-stats-method=#	对于NULL值的统计方式，可选的值有nulls_unequal、nulls_equal和nulls_ignored	nulls_unequal
--aria-sync-log-dir=#	可选的值有never、newfile和always	newfile

通常情况下，需要重点关注的几个参数有下面几个。

- aria-pagecache-bufer-size。所有的数据和索引都缓存在这个buffer中，该buffer越大Aria的运行速度就越快。
- aria-block-size。块大小，其默认值8192在大部分情况下都是可以的。通常情况下，块内查找使用的是二分查找，但在块内使用压缩存储键值的时候，在块内使用的是扫描查找，aria-block-size的值太大会影响块内的查询时间。可选的值为2048、4096和8192。

- aria-log-purge-type。如果你想保留事务日志文件并将其作为备份，请将该参数设置为af_flush。

3. Aria的PAGE行格式

MyISAM的DYNAMIC和FIXED行格式非常简单，而且不会占用过多的额外空间，当数据不会被修改的时候，它们表现得非常优秀。如果修改比较多，而且修改后的行数据比原来大，那么对于DYNAMIC行格式，性能会急剧下降。使用PAGE行格式，即使有大量的更新操作，也不会像DYNAMIC行格式那样，它仅会产生非常少的碎片。同时，PAGE行格式使用了页缓存，具有很好的随机性能。

4. Aria的优缺点

与MyISAM存储引擎相比，Aria存储引擎具有很多的优点，具体如下所示。

- Aria的数据和索引具有崩溃恢复功能，如果发生崩溃，Aria会回滚到命令执行前的状态。
- Aria能重放事务日志中的所有内容，因此你可以使用事务日志来备份Aria。但有一些不能重放：往空表批量插入（包括LOAD DATA INFILE、SELECT ... INSERT和INSERT多行），ALTER TABLE操作。
- LOAD INDEX能够跳过一些不需要的索引页。
- 除了支持所有MyISAM的行格式外，还支持页格式，数据存储在页内，页的默认大小为8KB。
- 支持对一张表的并发插入操作。
- 当使用页格式时，数据缓存在页缓存中。
- 同时支持崩溃恢复表类型（未来将实现事务）和非事务表类型。

相对于MyISAM存储引擎，Aria存储引擎也存在一些缺点，具体如下所示。

- Aria不支持延迟插入。
- 当使用页格式时，如果每一行的数据比较小（<25字节），Aria的性能会受影响。

在Aria的未来版本中，以上两个问题都将解决。

2.1.2　XtraDB存储引擎

XtraDB是目前MariaDB的默认存储引擎。XtraDB是由Percona公司基于InnoDB开发的高性能存储引擎，其主要目的是用于替代现有的InnoDB。可以将XtraDB看作是InnoDB的增强版本，它在InnoDB上进行了大量的修改，完全兼容现有的InnoDB，并且提供了许多InnoDB不具备的功能。

XtraDB对InnoDB主要做了以下一些优化。

- XtraDB在多核CPU上的性能和伸缩性更好。
- XtraDB对于内存的分配和使用更加合理和高效。

- ❑ 解除了InnoDB的许多限制。
- ❑ 提供了比InnoDB更多的配置和性能监控参数。

对于高负载的MariaDB/MySQL应用来说，完全可以使用XtraDB存储引擎来替代InnoDB存储引擎。

2.1.3 SphinxSE存储引擎

SphinxSE存储引擎主要用于全文检索，MariaDB从5.2开始将SphinxSE存储引擎加入进来。SphinxSE存储引擎是Sphinx项目的一部分，Sphinx是用C++编写的一款开源搜索引擎。虽然名字叫存储引擎，但SphinxSE不会真正地存储数据，它只是一个内置的客户端，让MariaDB/MySQL能够和Sphinx进行通信，能够在MariaDB/MySQL中执行查询语句，而Sphinx将匹配的结果返回来。所有的索引动作和搜索动作都发生在MariaDB/MySQL之外。

想要通过SphinxSE存储引擎进行搜索，你首先必须创建一个存储引擎为SphinxSE的表，然后在这个表上进行SELECT查询。

下面我们简单介绍一下如何使用SphinxSE。

首先通过CREATE TABLE创建一个表，指定存储引擎为Sphinx，并且指定Sphinx的searchd的位置以及使用哪个索引。搜索时只需要按照Sphinx的格式来执行SELECT语句就能返回想要的结果。相关代码如下：

```
CREATE TABLE t1
(
    id          INTEGER UNSIGNED NOT NULL,
    weight      INTEGER NOT NULL,
    query       VARCHAR(3072) NOT NULL,
    group_id    INTEGER,
    INDEX(query)
) ENGINE=SPHINX CONNECTION="sphinx://localhost:9312/test";

SELECT * FROM t1 WHERE query="test it;mode=any";
```

可以登录http://sphinxsearch.com查询与Sphinx相关的更加详细的信息。

2.1.4 FederatedX存储引擎

FederatedX存储引擎是Federated存储引擎的一个分支，其主要作用是访问远程的数据源。FederatedX存储引擎表并不存放实际的数据，它只是指向一台远程MySQL/MariaDB数据库服务器上的表。FederatedX存储引擎通过libmysql来访问远程的数据库。当前的FederatedX存储引擎只支持MySQL/MariaDB的表，不支持异构的数据库表。

2.1.5 TokuDB存储引擎

MariaDB从5.5开始加入了TokuDB存储引擎，该存储引擎使用了分形树作为内部索引的结构，在插入性能上表现优异。TokuDB非常适合插入性能要求特别高的应用场景。

2.1.6 Cassandra存储引擎

Cassandra存储引擎是一个NoSQL的存储引擎，MariaDB在10.0中引入了该引擎。MariaDB可以借助Cassandra存储引擎来访问Cassandra集群的数据，你可以通过多个MariaDB实例访问同样的Cassandra集群的数据，只要这些MariaDB的实例上都安装了Cassandra存储引擎。

Cassandra存储引擎的主要目的是为了将SQL和NoSQL的数据进行整合。Cassandra存储引擎提供了类似SQL的查询语句（CQL）来访问Cassandra集群中的数据。Cassandra存储引擎并没有让Cassandra成为一个SQL数据库，Cassandra存储引擎只是一个由SQL世界通往NoSQL世界的窗口，如图2-2所示。

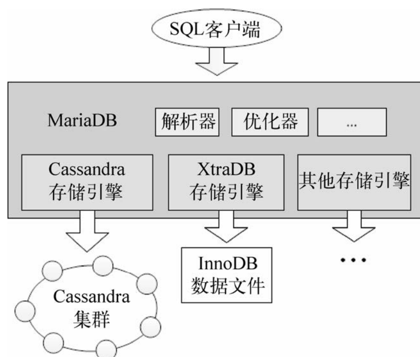


图2-2 Cassandra存储引擎

2.1.7 CONNECT存储引擎

有一部分数据以纯文本文件的形式或者其他的形式存储于关系数据库之外，而在许多企业中，这些数据需要在不同的系统中使用。通常的做法是通过ETL（Extract，Transform，Load）系统对这些数据进行处理，然后导入到关系数据库中供一些系统方便地使用。当这些外部数据的数据量非常大的时候，ETL系统需要花费很长的时间来处理这些数据。

上面描述的是外部数据的访问和管理问题。要是有一种方法能够轻松访问这些数据，和访问关系数据库中的数据那样，但是这些数据又不需要经过ETL之类的系统进行加工那该多好。因为我们知道，当外部数据量非常大时，ETL系统会消耗很长的时间来处理这些数据。CONNECT存储引擎在某些方面解决了这个问题。MariaDB从10.0开始引入了CONNECT存储引擎。它允许

MariaDB像访问数据库中表的数据那样访问外部数据。CONNECT存储引擎具有以下特点。

- 不需要添加额外的SQL语句扩展，通过CONNECT存储引擎可以用访问数据库内部表一样的方式访问外部数据。
- 内置了许多类型的外部数据的封装器，例如纯文本和一些数据源等。
- 能够通过CONNECT存储引擎对文本形式以及大多数数据源形式的外部数据进行读和写。
- CONNECT存储引擎只会访问需要的数据，不会访问多余的数据。
- 支持索引、特殊列和虚拟列。
- 对于分区表，能够实现并发执行。
- 能够在远程服务器上执行复杂查询。
- 提供了C++的接口让用户可以针对外部数据的类型定制封装器。

以上我们只是对CONNECT存储引擎进行了简单的介绍，有兴趣想要更深一步了解的读者可以自行查阅相关的资料。

2.1.8　Sequence存储引擎

Sequence存储引擎能够生成一个升序或者降序的整数序列，你可以指定这个序列的起始值和终止值以及序列增加的步长。Sequence存储引擎的表是虚拟的、短暂的，是不能持久化的，同时是只读的。Sequence存储引擎的表是在你执行SELECT语句时自动创建的，没有一种类似执行CREATE命令的方法来创建它们。创建一个Sequence存储引擎的表不会在磁盘上写入任何内容，也不会创建.frm格式的文件。MariaDB从10.0开始引进了Sequence存储引擎。

Sequence表的使用非常简单。当你想使用一组序列时，执行简单的SELECT语句就能返回你想要的序列，例如：

```
SELECT * FROM seq_1_to_5;
+-------+
|  seq  |
+-------+
|  1    |
|  2    |
|  3    |
|  4    |
|  5    |
+-------+
```

有两种形式的SELECT语句可以返回一组序列：select_FROM_to_TO_step_STEP和select_FROM_to_TO。上面的例子描述的是采用第二种形式的SELECT语句来生成序列，当我们不指定步长时，默认的步长是1。下面我们给出一个使用第一种形式指定步长生成序列的例子：

```
SELECT * FROM seq_5_to_1_step_2;
+-------+
| seq   |
+-------+
|  5    |
|  3    |
|  1    |
+-------+
```

Sequence的表虽然是虚拟的，但毕竟也是一种表，所以这个表必须属于某个库。在使用Sequence的表之前必须确保它已经处于某个库内，也就是说在之前已经执行过use命令。

2.1.9　Spider存储引擎

Spider存储引擎内置了分片功能，支持分区和分布式事务。通过Spider存储引擎，你可以像操作本地MariaDB实例的表一样来操作远程MariaDB实例上的表，也可以像操作本地MariaDB实例的表一样来操作分布在多个MariaDB实例上的表。

当创建一个Spider存储引擎的表时，该表指向远程服务器上对应的一张表或多个实例上的表，就像UNIX/Linux中的软链接一样。远程服务器上的表可以是任何存储引擎的表。

在执行CREATE TABLE命令创建Spider存储引擎的表时，需要添加COMMENT或CONNECTION语法来指定远程服务器的地址等信息。接下来，我们举两个例子来介绍一下Spider存储引擎的使用方法。第一个例子介绍了如何使用Spider存储引擎来访问远程实例上的数据，第二个例子介绍了如何使用Spider存储引擎对数据进行分片。

1. 利用Spider存储引擎访问远程实例的数据

例如，我们在远程服务器上存在一张表，创建表的语句如下：

```
node1 > CREATE TABLE s(id INT NOT NULL AUTO_INCREMENT, code VARCHAR(10), PRIMARY KEY(id));
```

在本地服务器上我们创建一个Spider类型的表：

```
CREATE TABLE s(id INT NOT NULL AUTO_INCREMENT, code VARCHAR(10), PRIMARY KEY(id))
ENGINE=SPIDER
COMMENT 'host "127.0.0.1", user "user1", password "pswd1", port "8607"';
```

可以将数据插入到本地的Spider表中，本质上该数据会被存储到远程服务器对应的表中：

```
INSERT INTO s(code) VALUES ('a');

node1 > SELECT * FROM s;
+-------+--------+
| id    |code    |
```

```
+-------+--------+
| 1     | a      |
+-------+--------+
```

2. 利用Spider存储引擎对数据进行分片

上面的例子描述了如何通过Spider存储引擎访问远程数据库中的表，下面我们给出一个通过Spider存储引擎对数据进行分片的例子。例如，我们有3个数据库节点，将使用其中的两个（DB1和DB2）作为存储数据的数据节点，另外一个作为Spider节点，供用户进行操作。

首先在两个数据节点上分别执行创建表语句，并且创建一个用户，让Spider节点可以通过该用户连接到数据节点上：

```
CREATE TABLE db_a.tbl_a(
    col_a int not null,
    col_b varchar(20),
    col_c int not null,
    primary key(col_a)
)ENGINE=INNODB;

CREATE USER user IDENTIFIED BY " password ";
GRANT ALL ON *.* TO user@"%" IDENTIFIED BY "password";
FLUSH PRIVILEGES;
```

然后在Spider节点的配置文件中加入以下选项：

```
spider_internal_xa=1
spider_semi_trx_isolation=3     #repeated read
```

接着在Spider节点上创建一个Spider表，并且让该表指向数据节点DB1和DB2：

```
CREATE TABLE db_a.tbl_a(
    col_a int not null,
    col_b varchar(20),
    col_c int not null,
    primary key(col_a)
)ENGINE=SPIDER CONNECTION='wrapper "mysql", user "user", password "password", table "tbl_a",
    port "3306"'
PARTITION BY KEY(col_a)
(PARTITION pt1 COMMENT='host "DB1"', PARTITION pt2 COMMENT='host "DB2"');
```

当我们访问Spider节点的db_a.tbl_a表时，Spider存储引擎自动将转向数据节点DB1和DB2，这样就很方便地实现了数据的分片功能，而且Spider存储引擎是支持分布式事务的，如图2-3所示。

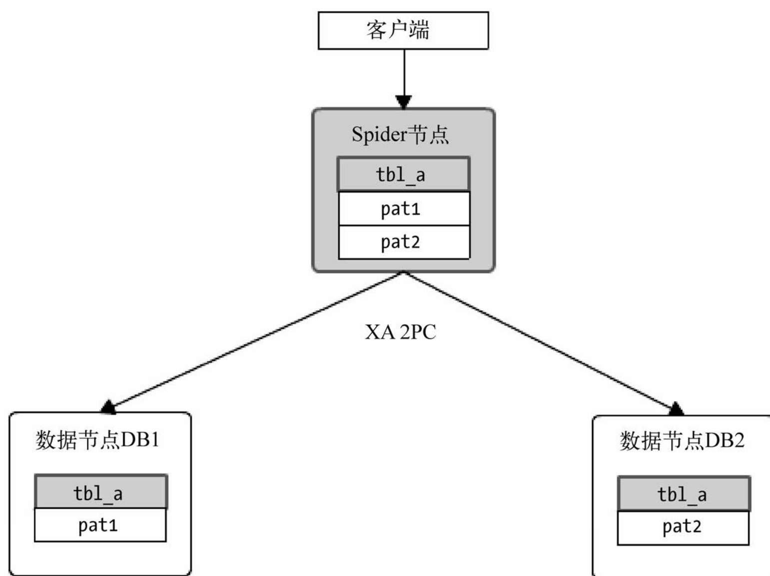


图2-3　Spider存储引擎

2.2　线程池技术和 binlog group commit 技术

MariaDB的线程池技术在解决最大连接数以及减少大量线程带来的系统开销问题上发挥了巨大的作用，而MariaDB的binlog group commit技术能够极大地提升数据库在大量并发事务下单位时间内的事务提交数。

2.2.1　线程池技术

我们知道，MySQL的传统连接模式是每连接每线程，MySQL会给每个连接上来的客户端分配一个单独的线程，该线程处理该客户端发来的所有命令。随着MySQL的连接数越来越多，MySQL的线程数也会相应地上升。MySQL默认的最大连接数是1024，也就是说当连接的客户端达到1024个之后，新来的客户端将连接不上MySQL的服务端。当然，可以通过调整参数`max_connections`来提高MySQL的最大连接数，但这又带来了其他问题：首先，每个线程会占用一定的系统资源，线程数越多消耗的系统资源也越多；其次，线程的创建和销毁是有一定开销的；最后，也是非常重要的一点，当线程数过多时，如果其中大部分线程都处于活跃状态，将会导致频繁的上下文切换，从而造成巨大的系统开销。

MariaDB从5.1开始引入了线程池技术来解决最大连接数限制问题以及过多线程带来的系统开销问题。线程池技术的本质就是线程复用，多个连接之间共享线程。在第5章中，我们将详细介绍MariaDB的线程池技术。

2.2.2　binlog group commit技术

我们知道，操作系统使用页面缓存机制来填补内存访问速度和磁盘访问速度之间的差距。通常情况下，对磁盘文件的写都会首先写入到页面缓存中，然后由操作系统来决定何时将修改过的脏页刷新到磁盘上。如果想确保修改已经持久化到了磁盘，必须调用fsync或者fdatasync。在关系数据库中，为了满足ACID中的D（持久化）属性，也就是说事务提交并且成功返回给客户端之后，必须保证该事务的所有修改不能丢。无论是在数据库程序崩溃的情况下，还是在数据库所在的服务器发生宕机或者断电的情况下，都必须保证数据不能丢，这就要求数据库在事务提交过程中调用fsync或fdatasync将数据持久化到磁盘。fsync是一个昂贵的系统调用，对于普通的磁盘，每秒只能完成几百次的fsync操作，很明显，fsync将会限制每秒钟提交的事务数，成为关系数据库的瓶颈。

对于MariaDB/MySQL，这种情况变得更加糟糕。在开启binlog的情况下，为了保证主库和从库之间数据的一致性，MariaDB/MySQL使用了事务的两阶段提交协议。在这种情况下，为了满足数据的持久化需求，一个事务的提交最多会导致3次fsync操作。

为了提高MariaDB/MySQL在开启binlog的情况下单位时间内的事务提交数，就必须减少每个事务提交过程中导致的fsync的调用次数。MariaDB从版本5.3开始，引入了binlog group commit技术来解决这个问题。MySQL从版本5.6开始也加入了binlog group commit技术，其他一些非官方的组织也提交了自己的补丁来解决这个问题，但基本思路都非常相似。

binlog group commit的基本思想是多个并发提交的事务之间共用一次fsync操作来实现事务对binlog修改的持久化。在第7章中，我们将对该技术做详细的分析。

2.3　MariaDB 其他扩展和新特性

除了上面介绍过的包含更多有用的存储引擎、使用线程池技术处理大量连接问题以及采用binlog group commit提高系统在高并发事务情况下的性能之外，MariaDB还包含了许多其他有用的特性，例如支持更精确的时间类型，支持虚拟列和动态列，提供更多的统计信息，支持某些命令执行进度报告，让用户实时了解命令的执行情况，等等。本节中，我们将简单介绍其中的一些特性，更加详细全面的介绍请参考MariaDB的官方文档https://mariadb.com/kb/en/mariadb-vs-mysql-features。

2.3.1　更高的时间精度

从MariaDB 5.3开始，TIME、DATETIME和TIMESTAMP这三个时间类型最高可以支持微秒级别的时间精度。在使用CREATE TABLE命令创建表时，可以为时间所在的列指定你想要的时间精度，示例如下：

```
CREATE TABLE example (
    col_microsec DATETIME(6),
    col_millisec TIME(3)
);
```

在上面的例子中，列col_microsec具有微秒级别的时间精度，而列col_millisec具有毫秒级别的时间精度。你可以在TIME、DATETIME和TIMESTAMP类型后面添加额外的用小括号括起来的整数来确定时间精度，整数的取值范围为0~6，如果在类型后面什么也没添加，是和添加 (0) 的效果一致，将使用秒级别的时间精度。

2.3.2　虚拟列

虚拟列是表中这样的列，它们的值是根据确定的表达式或者是根据表中其他列的值自动计算的。创建虚拟列的语法如下：

```
<type> [GENERATED ALWAYS] AS (expression)
[VIRTUL | PERSISTENT] [UNIQUE] [UNIQUE KEY] [COMMENT <text>]
```

其中type指定了列的类型，expression指定了计算虚拟列值的表达式。有两种虚拟列：被VIRTUAL关键字修饰的虚拟列，该列的值将会在查询的时候计算生成；被PERSISTENT关键字修饰的虚拟列，虚拟列的值存储在表中。下面给出了一个使用虚拟列的例子：

```
CREATE TABLE example_virtual_columns(
    a INT(11) PRIMARY KEY,
    b VARCHAR(32),
    c INT(11) AS (a mod 10) VIRTUAL,
    d VARCHAR(5) as (left(b, 5)) PERSISTENT);
```

虚拟列c的值将会在查询时计算，而虚拟列d的值被存储在表中，查询的时候直接从表里取出。当使用DESC命令查看表的定义时，可以很容易在Extra列中找到哪些列是虚拟列：

```
mysql> DESC example_virtual_columns;
+-------+------------------+-------+-------+-----------------+-------------------+
| Field | Type             | Null  | Key   | Default         | Extra             |
+-------+------------------+-------+-------+-----------------+-------------------+
| a     | int(11)          | NO    | PRI   | NULL            |                   |
| b     | varchar(32)      | YES   |       | NULL            |                   |
| c     | int(11)          | YES   |       | NULL            | VIRTUAL           |
| d     | varchar(5)       | YES   |       | NULL            | PERSISTENT        |
+-------+------------------+-------+-------+-----------------+-------------------+
```

同时很容易通过执行SHOW CREATE TABLE命令来查看生成虚拟列的表达式：

```
mysql> SHOW CREATE TABLE example_virtual_columns \G;
*************************** 1. row ***************************
Table: example_virtual_columns
```

```
Create Table: CREATE TABLE `example_virtual_columns` (
    `a` int(11) NOT NULL,
    `b` varchar(32) DEFAULT NULL,
    `c` int(11) AS (a mod 10) VIRTUAL,
    `d` varchar(5) AS (left(b, 5)) PERSISTENT,
    PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```


在执行插入操作时，虚拟列使用default关键字代替就可以。如果为虚拟列指定值，将会导致错误的发生：

```
mysql>INSERT INTO example_virtual_columns VALUES (16, "abcdefghijkl", default, default);
Query OK, 1 row affected (0.01 sec)
mysql> SELECT * FROM example_virtual_columns;
+-------+-------------------+--------+------------------+
| a     | b                 | c      | d                |
+-------+-------------------+--------+------------------+
| 16    | abcdefghijkl      |  6     | abcde            |
+-------+-------------------+--------+------------------+
1 row in set (0.00 sec)

mysql> INSERT INTO example_virtual_columns VALUES (17, "abcdefghijkl", default, "there");
ERROR 1906 (HY000): The value specified for computed column 'd' in table 'example_virtual_columns' ignored
```

2.3.3　用户统计功能

从版本5.2开始，MariaDB引入了用户统计（user statistics）功能，该功能主要用于统计客户端、使用者、索引使用情况以及表使用情况等信息，让用户更加了解自己的MariaDB实例的运行状况。

MariaDB的用户统计功能是基于Percona和Ourdelta的userstatv2包实现的，并且在此基础上做了许多改进，使其在性能上更快，在功能上更加全面。

1. 开启用户统计功能

默认情况下用户统计功能是关闭的，需要在MariaDB启动之前在配置文件或者命令行添加以下参数开启：

```
userstat = 1
```

此外，也可以执行SET命令动态启动该功能：

```
SET GLOBAL userstat = 1;
```

2. 查看用户统计信息

用户统计功能在INFORMATION_SCHEMA库中添加了CLIENT_STATISTICS、USER_STATISTICS、

INDEX_STATISTICS和TABLE_STATISTICS四个表，用来统计不同的信息。表CLIENT_STATISTICS记录了客户端连接相关的信息，例如客户端的IP地址、从某个IP地址过来的连接数等信息。表USER_STATISTICS记录的是用户行为相关的信息，例如哪个用户产生了大部分压力，哪个用户在滥用数据库资源等信息。表INDEX_STATISTICS统计索引的使用情况，通过查看该表的统计信息，可以发现一些没用的索引并将其删除，达到优化数据库的目的。表TABLE_STATISTICS记录表的使用情况，例如某个表修改的行数，从该表读取了多少行等信息。

有两种方式查询用户统计信息——使用SHOW或者SELECT命令：

```
SHOW table_name;
SELECT * FROM INFORMATION_SCHEMA.table_name;
```

其中*table_name*为4个表中的一个，例如以下命令用于查询客户端相关的统计信息：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.CLIENT_STATISTICS \G;
*************************** 1. row ***************************
                CLIENT: localhost
     TOTAL_CONNECTIONS: 3
CONCURRENT_CONNECTIONS: 0
        CONNECTED_TIME: 58887
             BUSY_TIME: 0.532569
              CPU_TIME: 0.027318300000000007
        BYTES_RECEIVED: 537
            BYTES_SENT: 21540
  BINLOG_BYTES_WRITTEN: 0
             ROWS_READ: 1
             ROWS_SENT: 17
          ROWS_DELETED: 0
         ROWS_INSERTED: 0
          ROWS_UPDATED: 0
       SELECT_COMMANDS: 8
       UPDATE_COMMANDS: 0
        OTHER_COMMANDS: 0
   COMMIT_TRANSACTIONS: 2
 ROLLBACK_TRANSACTIONS: 0
    DENIED_CONNECTIONS: 0
      LOST_CONNECTIONS: 0
         ACCESS_DENIED: 0
         EMPTY_QUERIES: 1
1 row in set (0.00 sec)
```

3. 重置用户统计信息

MariaDB提供了以下4个命令来重置相应的用户统计信息。通过执行这些命令，用户就可以在执行某些操作后重新开始监控MariaDB的运行状况：

```
FLUSH CLIENT_STATISTICS;
FLUSH USER_STATISTICS;
FLUSH INDEX_STATISTICS;
FLUSH TABLE_STATISTICS;
```


2.3.4 KILL命令的扩展

对于MySQL来说，在传统的每连接每线程模式下，你可以通过执行SHOW PROCESSLIST命令查看某条正在执行的语句对应的线程号，然后通过执行如下命令来终止一个连接或者终止该连接上正在执行的命令：

```
KILL [CONNECTION | QUERY] thread_id
```

当执行KILL命令时，如果携带了CONNECTION修饰词，将会终止*thread_id*指定的线程；如果携带了QUERY修饰词，仅仅终止该线程当前正在执行的命令，线程仍然保持；如果不带任何修饰词，其效果和携带CONNECTION修饰词一样，将会终止*thread_id*指定的线程。

MariaDB对KILL命令主要做了两个扩展，一个是添加了HARD|SOFT修饰词，另一个是终止某个用户的所有线程或正在执行的所有命令。MariaDB的KILL命令的语法如下：

```
KILL [HARD | SOFT] [CONNECTION | QUERY] [thread_id | USER  user_name | query_id]
```

如果在执行KILL命令时采用了HARD修饰词，那么将会尽快终止正在执行的命令或对应的线程；如果采用了SOFT修饰词，那么正在执行的某些命令（例如REPAIR或者CREATE INDEX命令）将不会被打断，因为这些命令如果被打断将会导致表处于不一致的状态。

KILL ... USER *user_name*将会终止*user_name*用户的所有连接或正在执行的所有命令，其中*user_name*可以通过以下3种方式指定：

- *user_name*
- *user_name* @localhost
- CURRENT_USER或者CURRENT_USER()

2.3.5 命令执行进度报告

在我们使用MySQL的过程中，可能会遇到在表有一定量的数据之后需要执行一些DDL操作来满足新增的需求。虽然我们提倡表的结构应该提前设计好，尽量不要在表的数据量很大时执行DDL操作，但你不能要求用户总能预测所有可能发生的需求。这样的DDL操作是非常耗时的，根据表的数据量的不同可能需要耗费数小时甚至数十小时。等待是个漫长的过程，特别是在不知道还要等多久的时候。

从版本5.3开始，MariaDB对某些耗时的命令提供了进度报告的功能，通过该功能，我们能够

清楚地了解到命令的执行进度，很容易估算出命令还需要持续多长时间。支持进度报告的命令包括以下几个：

- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- LOAD DATA INFILE

在执行SHOW PROCESSLIST命令时，会发现MariaDB比MySQL多了一列输出Progress，该列用来报告命令总的执行进度（0 ~ 100%）：

```
mysql> SHOW PROCESSLIST;
+-------+--------+-------------------------+---------+-----------------------+---------------+
| Id    | User   | Host                    | ...     | Info                  | Progress      |
+-------+--------+-------------------------+---------+-----------------------+---------------+
| 12    | root   | localhost:44097         | ...     | show processlist      | 0.000         |
+-------+--------+-------------------------+---------+-----------------------+---------------+
1 row in set (0.00 sec)
```

2.3.6　动态列

从版本5.3开始，MariaDB引入了动态列的概念，允许一个表的每一行存储不同列的信息。MariaDB的动态列通过将一个列集合存储在blob字段中，并且定义一个操作实现这些列的函数集合。动态列适合于某些不确定的场景，例如某个商品的属性个数不确定并且将来可能还会添加。

想要使用动态列，首先表中必须包含blob类型的列：

```
create table assets (
    item_name varchar(32) primary key,
    dynamic_cols  blob
);
```

接下来，就可以使用MariaDB定义的动态列操作函数对动态列进行存取操作：

```
INSERT INTO assets VALUES ("MariaDB T-shirt", COLUMN_CREATE("color", "blue", "size", "XL"));
INSERT INTO assets VALUES ("Thinkpad Laptop", COLUMN_CREATE ("color", "black", "price", 500));
```

以上两条语句往assets表中插入了两行记录，接下来查询商品的颜色情况：

```
mysql> SELECT item_name, COLUMN_GET(dynamic_cols, "color" as char) AS color FROM assets;
+----------------------------+--------+
| item_name                  | color  |
+----------------------------+--------+
| MariaDB T-shirt            | blue   |
| Thinkpad Laptop            | black  |
+----------------------------+--------+
```

此外，还可以动态删除或者增加某行的动态列：

```
UPDATE assets SET dynamic_cols = COLUMN_DELETE(dynamic_cols, "price") WHERE COLUMN_GET(dynamic_cols,
    "color" as char)= "black";                          --删除动态列

UPDATE assets SET dynamic_cols = COLUMN_ADD(dynamic_cols, "warranty", "3 years") WHERE
    item_name="Thinkpad Laptop";                        --增加动态列
```

你可以通过调用COLUMN_LIST函数来查看动态列的情况，或者使用COLUMN_JSON函数以JSON的格式来查看动态列以及它们对应的值：

```
SELECT item_name, COLUMN_LIST (dynamic_cols) FROM assets;
+-----------------------------+----------------------------------------+
| item_name                   | column_list(dynamic_cols)              |
+-----------------------------+----------------------------------------+
| MariaDB T-shirt             | "size","color"                         |
| Thinkpad Laptop             | "color","warranty"                     |
+-----------------------------+----------------------------------------+

SELECT item_name, COLUMN_JSON(dynamic_cols) FROM assets;
+-----------------------------+---------------------------------------------------------------+
| item_name                   | COLUMN_JSON(dynamic_cols)                                     |
+-----------------------------+---------------------------------------------------------------+
| MariaDB T-shirt             | {"size":"XL","color":"blue"}                                  |
| Thinkpad Laptop             | {"color":"black","warranty":"3 years"}                        |
+-----------------------------+---------------------------------------------------------------+
```

2.3.7 多源复制

MySQL能够轻松实现一主多从的功能，但是想将多个实例复制到一个实例中还是比较难的。当然，可以通过多级复制来达到目的，但这种方式不是那么优雅，而且存在一定的问题，例如网络流量的增加以及产生冗余的binlog等。幸好MariaDB很好地解决了这个棘手的问题。从10.0开始，MariaDB引入了多源复制的机制，允许一个从库可以指定多个主库作为数据源。多源复制允许将多个实例的数据进行聚合，然后进行备份或者进一步的分析。

我们将在第8章中对MariaDB的多源复制进行详细深入的介绍。

2.4 小结

本章中，我们对MariaDB的一些扩展以及新特性进行了简单的介绍，这里我们简单回顾这些内容。

MariaDB相对于MySQL具有更多的存储引擎，用于满足不同的需求。同时，MariaDB还拥有自己特有的存储引擎——Aria存储引擎。Aria存储引擎支持崩溃恢复功能，相对于MyISAM来说

具有更好的数据安全性。

MariaDB的线程池技术在解决最大连接数限制问题以及过多线程带来的系统开销问题方面发挥了巨大作用。MariaDB的binlog group commit技术通过在并发的多个事务之间共享fsync/fdatasync操作来使MariaDB在高并发事务下得到极大的性能提升。MariaDB还支持其他一些非常有用的特性，例如支持动态列、虚拟列和多源复制，等等。

第 3 章

初识MariaDB源代码

开源一个很大的好处就是源代码可以很容易获取，有了源代码我们就可以阅读学习或者进行二次开发。MariaDB的源代码相对来说比较多，这让许多有兴趣阅读它的读者不知道从哪儿开始，以致望而却步。本章中，我们首先来讲解MariaDB源代码的目录组织结构，让读者对MariaDB的源代码有个宏观上的认识，然后介绍MariaDB对基础类型以及函数的封装，最后介绍如何调试MariaDB。读者可以按照程序的执行流来阅读MariaDB的代码。

本章的内容主要包括：

- MariaDB源代码的目录组织结构
- MariaDB对类型和函数的封装
- 调试MariaDB

3.1 MariaDB 源代码的目录组织结构

获得MariaDB的源代码包之后，采用适当的解压命令将其解压，得到的源代码目录结构如图3-1所示。

```
jinpengzhang@jinpengzhang:~/mariadb-10.0.6$ ls
BUILD            CREDITS              libmysql     randgen        TODO
BUILD-CMAKE      dbug                 libmysqld    README         unittest
client           debian               libservices  scripts        VERSION
cmake            Docs                 man          sql            vio
CMakeLists.txt   extra                mysql-test   sql-bench      win
cmd-line-utils   include              mysys        sql-common     zlib
config.h.cmake   INSTALL-SOURCE       mysys_ssl    storage
configure.cmake  INSTALL-WIN-SOURCE   packaging    strings
COPYING          KNOWN_BUGS.txt       pcre         support-files
COPYING.LESSER   libevent             plugin       tests
```

图3-1　MariaDB源代码的目录结构

表3-1以列表的形式将MariaDB源代码主要目录的作用列举出来。MariaDB作为MySQL的一个分支，代码的组织结构和MySQL有很多类似的地方，所以列表中的内容对于想要阅读MySQL源代码的读者同样具有一定的参考价值。表3-1中列出的是MariaDB 10.0.6的源代码结构，新版本的

代码结构可能会有一些改动，但大部分的结构会比较稳定。

表3-1　MariaDB源代码的目录结构

目录名称	说　明
BUILD	开发者构建脚本
client	命令行客户端程序代码
cmake	cmake使用的文件夹
cmd-line-utils	命令行客户端的外部库（libedit和readline）
dbug	调试库
Docs	模块的说明文档
extra	其他工具代码，包括jemalloc和yaSSL
include	包含文件，包括基础类型和数据结构的定义等
libevent	libevent网络库的源代码
libmysql	客户端连接服务器相关的代码
libmysqld	以STANDALONE模式使用MariaDB的功能，不需要通过客户端连接到MariaDB
man	UNIX手册页
mysql-test	回归测试包
mysys	基础接口的可移植性封装，例如文件读写接口、锁封装，等等
mysys_ssl	对OpenSSL/yaSSL的封装
packaging	打包工具
pcre	PCRE模式匹配库的相关代码
plugin	MariaDB的某些插件的代码，例如半同步复制插件相关的代码
randgen	生成测试语句的脚本
scripts	各种用途的脚本，例如在我们安装完MariaDB后需要执行该目录下的mysql_install_db.sh脚本创建一些基本的数据库
sql	服务端的所有核心代码。mysqld的启动main函数在该文件夹下的mysqld.cc文件中
sql-bench	压力测试的相关脚本
sql-common	客户端和服务端都会用到的一些代码
storage	包含了所有的存储引擎相关的代码
strings	自定义字符串库相关的代码
support-files	各种配置文件实例和实用脚本
tests	一些难以重现bug的相关测试
unittest	核心API的单元测试
vio	底层可移植网络I/O的相关代码
win	Windows平台下的打包升级工具
zlib	zlib压缩库的相关代码

3.2 MariaDB 对类型和函数的封装

MariaDB对基本的类型以及一些系统调用和C函数进行了封装，以达到更好的移植性和更好的抽象。

3.2.1 对类型的封装

在C/C++程序中，我们经常使用typedef关键字来定义类型别名，或者使用typedef简化复杂的类型声明，示例如下：

```
/*=============example 3-1 : typedef定义类型别名==============*/

// 使用原有类型声明
int socket_fd;

// 使用类型别名声明
typedef int SOCKET_T;
SOCKET_T socket_fd;

/*=============example 3-2 : typedef简化复杂的声明=============*/

int *(*a[5])(int, char*);          // 声明一个函数指针数组a

typedef int *(*pFun)(int, char*);   // 使用typedef简化a的声明
pFun a[5];
```

在第一个例子中，我们定义了int的别名SOCKET_T，这样在我们使用套接字的时候，使用的是类型SOCKET_T，而不是int。一方面，类型别名使程序具有更好的易读性，表达的意思更加确切；另一方面，别名为程序提供更好的抽象，当某一天套接字描述符不再是int类型时，我们只需要更改typedef的定义就可以了。

下面是MariaDB中几个类型别名的例子，大部分的类型别名都以my_开头：

```
typedef int my_socket;
typedef int File;
typedef unsigned long long int my_off_t;
typedef char my_bool;
```

除了使用typedef为基本类型定义类型别名外，MariaDB还对一些经常使用并且与系统相关的类型进行了封装，主要目的是为了实现跨平台。例如，mysql_mutex_t是对类型pthread_mutex_t的封装，mysql_cond_t是对类型pthread_cond_t的封装。

3.2.2 对函数的封装

MariaDB对系统调用以及一些C的库函数进行了封装。大部分的封装函数都是以my_开头的，

例如my_malloc、my_realloc和my_free是对C的库函数malloc、realloc和free的封装，my_open和my_close是对系统调用open和close的封装。

MariaDB对函数进行封装的目的主要有以下几个。

- **跨平台**。在不同的操作系统上，同样功能的函数可能名称并不相同，甚至在某些操作系统上没有相对应的函数，为了使代码在不同的操作系统上都能够正常运行，需要对这些函数进行封装。
- **执行额外的操作**。例如，MariaDB对C的库函数malloc进行了封装，而my_malloc函数比malloc函数多了一个my_flags参数，通过该参数可以指定分配内存失败时的额外行为，例如打印出现错误信息：

```
void *malloc(size_t size);
void *my_malloc(size_t size, myf my_flags);
```

- **更好的抽象**。想象一下这种情况，如果我们在程序中直接使用系统提供的函数，在程序的各个地方充斥着这些函数调用，当某一天发现有另一个函数B具有和函数A相同的功能，而且性能更好，这个时候如果想要使用函数B来代替函数A，就需要将所有出现函数A的地方用函数B来取代。但是如果根据函数的功能进行了抽象，然后进行相应的封装，这样我们的程序仅仅和抽象的概念相关，而没有和具体的实现或者某个函数相耦合。

3.3　调试 MariaDB

MariaDB/MySQL的源代码非常多，以至于许多读者不知道该从哪里开始阅读。一种很好的方式就是对MariaDB进行调试，跟踪MariaDB的启动过程以及命令的执行过程，顺着程序的执行流来阅读MariaDB的源代码。

3.3.1　准备工作

在对MariaDB进行调试之前，我们需要做一些准备工作，例如，MariaDB必须编译成Debug版本，我们必须熟悉GDB的常用命令，等等。

1. 编译MariaDB的Debug版本

在获取MariaDB的源代码之后，我们需要对其进行编译安装，然后才能运行。如果想要对其进行调试，就需要把MariaDB编译成Debug版本，这需要在执行cmake命令进行编译时携带DWITH_DEBUG参数：

```
root#  cmake . -DWITH_DEBUG ...
```

2. GDB的使用技巧

工欲善其事，必先利其器，好的工具总是能使我们的工作达到事半功倍的效果。

GDB（GNU Project Debuger）是GNU开源组织发布的一个强大的UNIX下的程序调试工具，使用它能够清楚地了解到其他程序正在做什么事情。通常，我们主要使用GDB来跟踪程序的执行过程，观察程序中相关变量的值，定位程序中存在的问题，或者使用GDB来调试coredump文件，定位导致程序崩溃的原因。

接下来我们介绍GDB的一些使用技巧。下面给出了`gdb --help`命令输出提示信息，其中列出了GDB的使用方法和各个选项的含义：


```
jinpengzhang@jinpengzhang:~$ gdb --help
This is the GNU debugger.  Usage:

    gdb [options] [executable-file [core-file or process-id]]
    gdb [options] --args executable-file [inferior-arguments ...]

Options:

  --args             Arguments after executable-file are passed to inferior.
  -b BAUDRATE        Set serial port baud rate used for remote debugging.
  --batch            Exit after processing options.
  --batch-silent As for --batch, but suppress all gdb stdout output.
  --return-child-result
                     GDB exit code will be the child's exit code.
  --cd=DIR           Change current directory to DIR.
  --command=FILE, -x  Execute GDB commands from FILE.
  --eval-command=COMMAND, -ex
                     Execute a single GDB command.
                     May be used multiple times and in conjunction
                     with --command.
  --core=COREFILE    Analyze the core dump COREFILE.
  --pid=PID          Attach to running process PID.
  --dbx              DBX compatibility mode.
  --directory=DIR    Search for source files in DIR.
  --epoch            Output information used by epoch emacs-GDB interface.
  --exec=EXECFILE    Use EXECFILE as the executable.
  --fullname         Output information used by emacs-GDB interface.
  --help             Print this message.
  --interpreter=INTERP
                     Select a specific interpreter / user interface.
  -l TIMEOUT         Set timeout in seconds for remote debugging.
  --nw               Do not use a window interface.
  --nx               Do not read gdbinit file.
  --quiet            Do not print version number on startup.
  --readnow          Fully read symbol files on first access.
```

```
  --se=FILE          Use FILE as symbol file and executable file.
  --symbols=SYMFILE  Read symbols from SYMFILE.
  --tty=TT           Use TTY for input/output by the program being debugged.
  --tui              Use a terminal user interface.
  --version          Print version information and then exit.
  -w                 Use a window interface.
  --write            Set writing into executable and core files.
  --xdb              XDB compatibility mode.

At startup, GDB reads the following init files and executes their commands:
   * system-wide init file: /etc/gdb/gdbinit

For more information, type "help" from within GDB, or consult the
GDB manual (available as on-line info or a printed manual).
Report bugs to "<http://bugs.launchpad.net/gdb-linaro/>".
```

使用GDB的方式比较多，主要有以下3种。

- 使用GDB调试正在运行的程序：

```
gdb -p pid
```

- 使用GDB启动程序进行调试：

```
gdb executable-file
```

- 使用GDB调试coredump文件：

```
gdb -c coredumpfile
```

当通过以上方式进入GDB命令行之后，就可以执行GDB命令来跟踪程序。表3-2给出了常用的几个GDB命令及其作用。

表3-2　GDB常用命令

命　令	缩　写	说　明
breakpoint	b	设置断点。可以通过指定函数名称或者哪个文件的行数来指定断点的位置
continue	c	继续执行到下一个断点或者直到程序结束
next	n	执行下一步，不进入子函数
step	s	执行下一步，进入子函数
backtrace	bt	打印当前的堆栈信息
info breakpoint	info b	查看所有断点信息
print	p	打印变量的值
info threads	i th	列出所有正在运行的线程的信息
thread	thr	切换到指定的线程
layout src	/	在控制台划出一个区域展示源代码，该命令在调试过程中阅读源代码非常有用

3.3.2　mysqld关键的函数调用

mysqld是MariaDB/MySQL的服务端程序，如果mysqld处于运行状态，那么用户就可以使用客户端连接并进行操作。

mysqld启动之后，首先进行必要的初始化工作，例如解析配置文件、初始化日志、加载必要的插件、启动slave线程（如果当前库是从库），等等，然后等待客户端的连接。

图3-2给出了mysqld在处理客户端连接时的相关函数调用。在调试MariaDB的过程中，我们可以选择性地在这些地方设置断点，以便跟踪命令的执行过程。

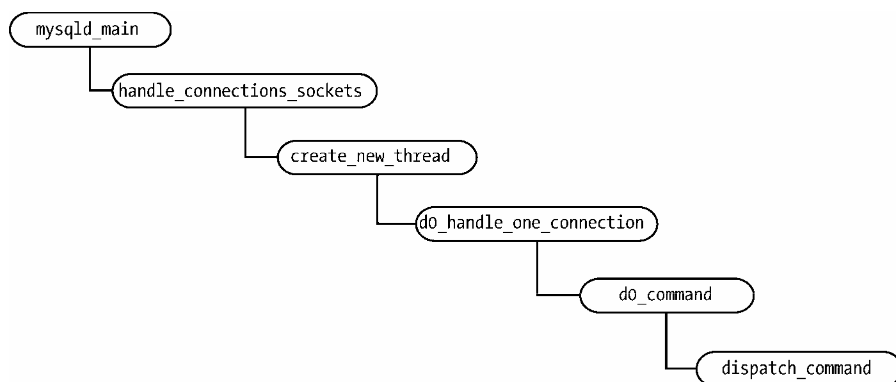


图3-2　mysqld函数调用

下面我们介绍一下这些函数的定义以及主要作用。

1. mysqld_main

该函数是mysqld启动的入口函数，所有的工作都从这里开始，包括配置文件的读取、命令行的解析、加载必要的插件，等等，其声明如下：

```
// sql/mysqld.cc

int mysqld_main(int argc, char **argv);
```

2. handle_connections_sockets

该函数在`mysqld_main`函数完成所有的初始化工作之后被调用，主要用来处理来自客户端的连接。如果采用的是每连接每线程的模式，那么当新的客户端连接到达时，该函数会为该连接创建一个单独的线程。该函数的声明如下：

```
// sql/mysqld.cc

void handle_connections_sockets();
```

3. create_new_thread

该函数的主要功能是调用系统相关的线程函数创建新的线程来处理新来的客户端连接，并且增加连接的计数，其声明如下：

```
// sqlmysqld.cc
void create_new_thread(THD *thd)
```

4. do_handle_one_connection

在每连接每线程模式下，会创建单独的线程来处理新来的连接，新创建的线程会进入do_handle_one_connection函数执行。该函数的声明如下：

```
// sql/connect.cc

void do_handle_one_connection(THD *thd);
```

5. do_command

该函数从客户端连接中读取一条命令，然后调用dispach_command函数执行该命令。该函数的声明如下：

```
// sql/sql_parse.cc

bool do_command(THD *thd);
```

6. dispatch_command

该函数用于执行从客户端读取的命令，其声明如下：

```
// sql/sql_parse.cc

bool dispatch_command(enum enum_server_command command, THD *thd,char *packet, uint packet_length);
```

其中参数command为命令的类型，参数thd为当前连接的上下文信息，参数packet和packet_ length指定了读取命令的详细信息。

3.3.3　调试

做好以上的准备工作后，接下来就可以对MariaDB/MySQL服务端程序进行调试，具体步骤如下。

(1) MariaDB必须编译成Debug版本，前面我们已经进行了相关的介绍。
(2) 启动MariaDB，使用客户端连接上来。
(3) 使用GDB调试mysqld进程，具体步骤如下。

(a) 首先通过ps命令查看mysqld的进程号，如图3-3所示。

```
root# ps -ef | grep mysqld
```

```
root@jinpengzhang:/usr/local/mariadb10/bin# ps -ef | grep mysqld
root     22189 22019  0 16:20 pts/6    00:00:00 /bin/sh ./mysqld_safe --defaults
-file=/etc/mariadb_my.cnf
mysql    22403 22189  1 16:20 pts/6    00:00:00 /usr/local/mariadb10/bin/mysqld
--defaults-file=/etc/mariadb_my.cnf --basedir=/usr/local/mariadb10 --datadir=/us
r/local/mariadb10/data --plugin-dir=/usr/local/mariadb10/lib/plugin --user=mysql
 --log-error=/usr/local/mariadb10/data/jinpengzhang.err --pid-file=jinpengzhang.
pid --port=3308
```

图3-3 查看mysqld的进程号

(b) 从图3-3中我们知道mysqld的进程号为22403，然后使用gdb命令进入mysqld程序（注意，必须具有root权限）：

```
root# gdb -p 22403
```

(c) 设置相应的断点。如图3-4所示，我们在dispatch_command函数处设置了相应的断点，这是命令执行的入口函数。

```
(gdb) b dispatch_command
Breakpoint 1 at 0x6311ad: file /home/jinpengzhang/jinpeng/mariadb-10.0.6/sql/sql
_parse.cc, line 1094.
(gdb)
```

图3-4 设置断点

(d) 通过已连接的客户端向MariaDB发送命令。如图3-5所示，我们发现客户端卡在这里，因为mysqld已经命中了我们设置的断点。

```
MariaDB [test]> insert into tb values(2008, "BeiJing", 1);
```

图3-5 客户端发送命令

(e) 在GDB中使用layout src命令划分一个窗口展示代码，我们可以直接在这个窗口中阅读相关的代码。接下来使用next命令单步执行，如果想看某个函数的具体实现，可以使用step命令进入该函数，如图3-6所示。

```
   1089        1   request of thread shutdown, i. e. if command is
   1090            COM_QUIT/COM_SHUTDOWN
   1091    */
   1092    bool dispatch_command(enum enum_server_command command, THD *thd,
   1093                          char* packet, uint packet_length)
B+>1094    {
   1095      NET *net= &thd->net;
   1096      bool error= 0;
   1097      DBUG_ENTER("dispatch_command");
   1098      DBUG_PRINT("info", ("command: %d", command));
   1099
   1100    #if defined(ENABLED_PROFILING)
   1101      thd->profiling.start_new_query();
multi-thre Thread 0x7f436 In: dispatch_command              Line: 1094 PC:
(gdb)
```

图3-6 单步调试

3.4　小结

本章中，我们首先介绍了MariaDB源代码的目录结构，让读者对MariaDB的源代码有一个宏观的认识，然后介绍了MariaDB对类型以及函数的封装情况，最后介绍了调试MariaDB的一些技巧。

第 4 章

MariaDB基础数据结构

本章中我们将对MariaDB中经常使用的几个底层数据结构进行分析，这些数据结构的存在是为了解决不同的问题。例如，MEM_ROOT的主要目的是为了加快动态内存的分配速度以及减少程序中内存碎片的发生；IO_CACHE作为文件读写的缓存，目的是为了提高文件读写的效率；NET封装了所有底层的网络操作，使上层网络相关的代码简单易懂，具有较高的移植性；THD包含了线程的所有上下文信息，MariaDB中很多函数的第一个参数都是THD类型的；一个TABLE_SHARE实例对应于数据库中的一个表，TABLE_SHARE保存了表的一些基本信息，例如字段名称、字段类型等；当我们执行查询语句的时候，需要打开语句中指定的表，这时就会创建一个TABLE对象来对应打开的表。这些基础的数据结构频繁地出现在MariaDB的很多地方，接下来我们对它们进行详细的分析，帮助想要了解MariaDB底层机制和想自行阅读MariaDB源代码的读者打下良好的基础。

本章的内容主要包括：

- 内存池MEM_ROOT
- 文件缓存IO_CACHE
- NET结构
- 线程上下文——THD
- TABLE_SHARE
- TABLE

4.1 内存池 MEM_ROOT

在C/C++中，当我们的程序需要动态分配内存的时候，就会调用malloc函数向内存分配器请求所需的内存，在结束内存的使用之后调用free函数将内存归还给内存分配器。

MariaDB采用MEM_ROOT内存池来管理动态内存的分配。MEM_ROOT向内存分配器请求大块连续的内存，程序向MEM_ROOT请求所需的动态内存。这种方式的内存分配策略一方面减少了分配内存时malloc的调用次数，提高了内存分配的速度，另一方面在一定程度上减少了程序中内存碎片的发生。

4.1.1　内存碎片问题

内存碎片一直是内存管理过程中一个棘手的问题，它不能完全避免，只能通过一些策略来减少发生，例如Linux内核使用伙伴系统和slab对象缓存来减少内存碎片的发生。

当程序需要内存的时候会向内存管理器提出内存申请，程序使用完内存后会将其归还给内存管理器，以供后续使用。程序每次请求的内存块的大小以及什么时候释放所请求的内存都各不相同，在程序的生命周期内，会有多次请求和释放内存，空闲内存开始是大块连续的，随着程序的运行以及内存的分配与释放，大块连续的空闲内存被分割成不连续的小块的空闲内存，当程序再次请求分配大块连续的内存时，虽然此时空闲内存的总量大于所请求的内存大小，但是由于没有连续大块的内存存在，导致请求得不到满足，这就是内存碎片问题。内存碎片通常分为内部碎片和外部碎片两种。

1. 内部碎片

内部碎片是指已经被分配的却不能被利用的内存空间。通常内存分配器实际分配给程序的内存经常比所请求的更大。内存分配器分配给程序的内存通常是4、8或者16的倍数。例如，在64位操作系统上的glibc内存分配器会对程序请求的内存按照16进行对齐，当程序请求23字节的时候，经过对齐操作之后，实际分配给该程序的是32字节，这种情况发生时多分配的9字节将会被浪费。同时为了在调用free函数释放已分配的内存块时能够进行正确的操作，内存分配器还会分配额外的空间记录内存块的大小。

当程序内部包含很多被分配了但是没有使用的内存时，下一次请求一大块连续的内存可能会失败，而实际上，程序内部未使用的这些碎片内存的大小总和可能比所请求的内存更大，这种情况叫作内部碎片。

2. 外部碎片

外部碎片是指还没有被分配出去的，但由于太小而无法满足程序的大块连续内存申请的空闲内存。当大块连续的空闲内存被已经分配了的内存块切分成小块不连续的内存时，虽然还有很多空闲的内存，但满足不了程序大块连续的内存请求。例如，某个程序请求了3个连续的内存块A、B和C，当程序将B块内存释放后，内存B可以用来满足小于等于B块大小的内存请求，却不能满足大于B块大小的内存请求。外部碎片同样存在于文件系统中，例如随着文件的创建、修改以及删除，磁盘中间会出现很多未被利用的碎片。

3. 伙伴系统

在C/C++程序中调用malloc函数申请内存时，分配的仅仅是虚拟地址，这个时候并没有分配对应的物理页框，只有当我们操作已分配的内存时触发缺页中断，内核才会分配相应的物理页框。Linux内核在管理物理内存页框的时候，也会遇到我们上面提到的外部碎片问题，频繁地请求和

释放不同大小的一组连续的物理页框，会导致本来连续的大块空闲页框被分割为不连续的小块空闲页框，由此导致即使有很多空闲的物理页框，但是满足不了一个大块的连续页框的请求。

Linux内核通过伙伴系统（buddy system）算法解决这种外部碎片问题。内核将所有空闲的连续页框分为11个组，每个组对应一个空闲块链表，链表中的空闲块分别包含1、2、4、8、16、32、64、128、256、512、1024个连续的物理页框，每个空闲块的第一个页框的物理地址是该块大小的整数倍，例如大小为32个连续页框的空闲块的起始地址是$32*2^{12}$的倍数（页的大小为4KB）。假设请求一个包含256个连续页框的空闲块，算法会先在256对应的链表中查找，看看是否存在这样的空闲块，如果有则直接从该链表中分配，如果没有，那么会向更大一级别的链表中查找，也就是512对应的链表找一个空闲的块，将其分割为两半，一半用于满足请求，另一半包含256个连续页框的块被加入256对应的空闲链表中。如果在512对应的链表中也没有找到空闲的内存块，那么继续找更大的块，也就是包含1024个连续页框的块，如果这样的块存在，内核把其中的256个连续的页框用于满足申请，剩余的768个连续的页框被分割为两块，一个包含256个连续页框的块和一个包含512个连续页框的块，这两个空闲块分别被加入到256和512对应的空闲链表中，如图4-1所示。

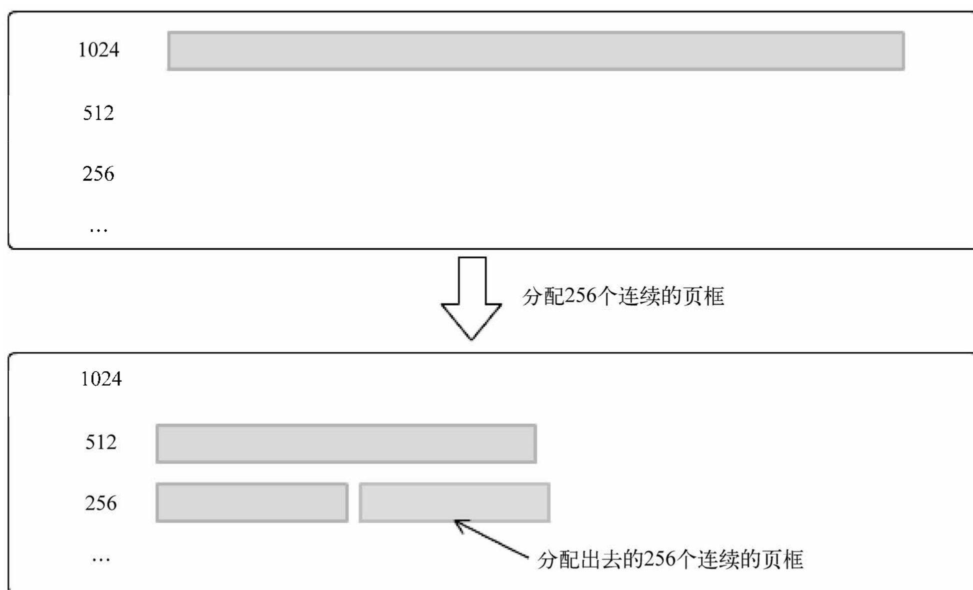


图4-1 伙伴系统分配物理页框

页框的释放过程并非简单地将块加入对应的空闲链表中，内核试图把大小同样为b的空闲伙伴块进行合并。所谓伙伴块，是指具有大小一致（同为b），并且它们的物理地址是连续的，同时第一个块的第一个页框的物理地址必须为$2*b*2^{12}$的倍数。该算法的合并过程是迭代的，当合并为大小为2b的空闲块之后，会试着寻找大小为2b的块的空闲伙伴块进行合并，以便形成更大的空闲

块，直到不能进行合并时，将合并完的块加入到对应的空闲链表中，如图4-2所示。

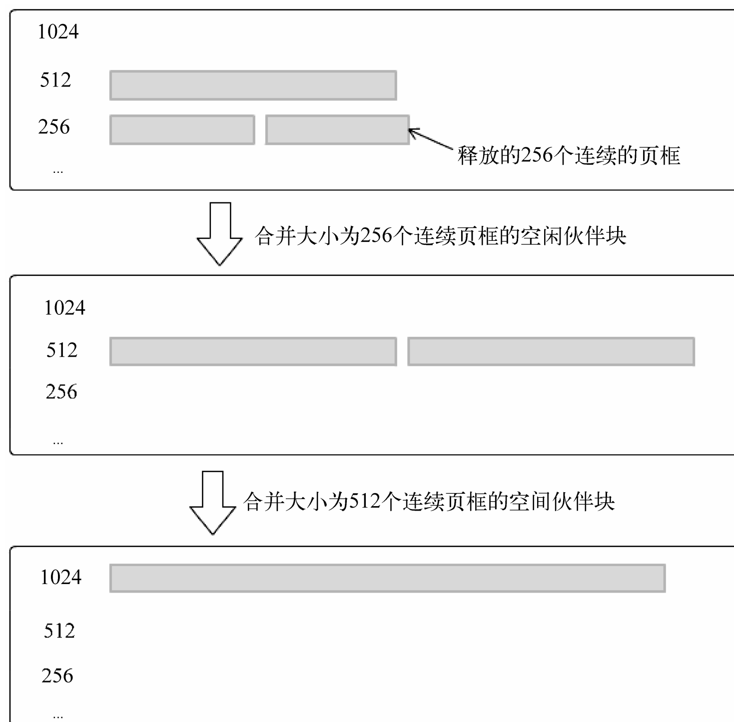


图4-2　伙伴系统合并空闲伙伴块

伙伴系统算法不仅存在Linux内核中，glibc的内存管理器（ptmalloc）同样采用了伙伴系统算法来管理空闲的内存。

4.1.2　MEM_ROOT 的定义

通过上面的介绍，我们知道伙伴系统能够很好地解决外部碎片问题，而在进程内通常都是内部碎片问题，为了避免内部碎片的产生，一个很好的策略就是总是向内存管理器请求4、8或者16倍数大小的大块内存，减少非对齐小块内存的申请。MEM_ROOT就采用了这样的思路对内存进行管理，MEM_ROOT向内存分配器申请大块对齐的内存，应用程序向MEM_ROOT申请小块的内存。

图4-3给出了MEM_ROOT的结构。USED_MEM结构代表从内存分配器分配的一个内存块，其中包括了该内存块的大小信息以及该内存块的使用情况。MEM_ROOT管理所有已分配的内存块，free链表中包含了有足够大的剩余空间能够继续满足内存分配请求的内存块，used链表中包含了具有很小的剩余空间甚至全部用满的内存块。

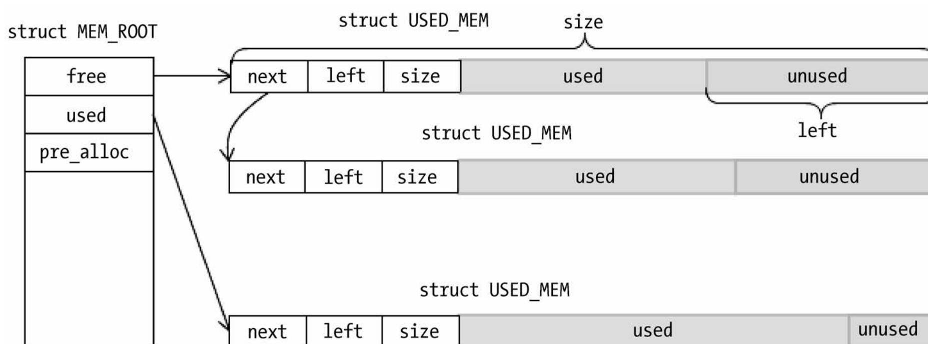


图4-3 MEM_ROOT结构图

下面我们给出USED_MEM和MEM_ROOT的定义：

```
// include/my_alloc.h

typedef struct st_used_mem
{
    struct st_used_mem *next;         // 下一个块
    unsigned int left;                // 当前块中剩余的字节数
    unsigned int size;                // 当前块的大小
} USED_MEM;

typedef struct st_mem_root
{
    USED_MEM *free;                   // 空闲内存块链表
    USED_MEM *used;                   // 已用内存块链表
    USED_MEM *pre_alloc;              // 初始化时预分配的内存块

    size_t min_malloc;                // 当空闲块的可用内存低于该值时将其移动到已用内存块链表中
    size_t block_size;                // 分配块的大小
    unsigned int block_num;           // 用于计算新分配块的大小

    unsigned int first_block_usage;

    void (*error_handler)(void);      // 错误处理函数
} MEM_ROOT;
```

USED_MEM中各个成员的含义如下。

- next：指向下一个内存块，用于实现链表。
- left：当前内存块还剩下多少字节可以使用，对应于图4-3中unused指示的区域。
- size：当前内存块的大小，包括USED_MEM结构的大小，如图4-3所示。

MEM_ROOT中各个成员的含义如下。

- free：空闲内存块链表。空闲内存块链表中的内存块不是没有被使用，而是这些块剩余的可用内存比较多，能够继续满足后续的内存申请。
- used：已用内存块链表。当内存块中剩余的字节数小于某个阈值时，会被移动到该链表中，虽然这样的策略可能会造成一定的浪费，但是能够快速地满足内存分配请求，因为该内存块剩余的空闲内存只能满足小于等于阈值的内存请求，如果仍然将它留在空闲内存块链表中，当这样的内存块在空闲块链表中积累时，势必会降低大于阈值的内存请求的效率。我们在分析MEM_ROOT内存分配的时候会进一步讲解。
- pre_alloc：初始化时预分配的内存块。
- min_malloc：当空闲链表中某个内存块的可用内存低于min_malloc指定的字节数时，这个内存块会被移动到已用内存块链表。
- block_size：新分配的内存块大小是block_size的整数倍。
- block_num：当需要分配新的内存块时，用于计算新分配内存块的大小。
- first_block_usage：计数器。当空闲链表中的第一个内存块多次不能满足内存申请时，出于效率的考虑，会将其加入到已用内存块链表中。
- err_handler：错误处理函数。

4.1.3　MEM_ROOT 的使用

上面给出了MEM_ROOT的相关定义，下面我们给出几个使用MEM_ROOT的例子来加深对它的理解。MEM_ROOT的使用非常简单，首先对其结构进行初始化，然后就可以使用alloc_root函数向它请求内存，最后需要调用free_root函数释放所有的内存块。如果以MY_MARK_BLOCKS_FREE标志调用free_root函数，表示仅仅将MEM_ROOT中的内存块标记为空闲状态，并不将其返回给内存分配器，这样这些内存块就能重复使用。示例代码如下：

```
/*=============example 4-1：MEM_ROOT的使用==============*/

MEM_ROOT mem_root;
init_alloc_root(&mem_root, 4096, 0);            // 初始化MEM_ROOT

char *p1 = (char*)alloc_root(&mem_root, 128);   // 分配内存
char *p2 = (char*)alloc_root(&mem_root, 64);
...

free_root(&mem_root, 0);                        // 释放内存

/*=============example 4-2：MEM_ROOT的使用==============*/

MEM_ROOT mem_root;
init_alloc_root(&mem_root, 4096, 4096);         // 初始化MEM_ROOT，预先分配一个内存块

char *p1 = (char*)alloc_root(&mem_root, 128);   // 分配内存
```

```
char *p2 = (char*)alloc_root(&mem_root, 64);
...

free_root(&mem_root, MY_MARK_BLOCKS_FREE);      // 释放内存，仅仅将所有已分配的内存块标记为空闲
```

4.1.4 MEM_ROOT 的初始化

从上面的例子中可以看出，MEM_ROOT不能直接使用，必须先调用init_alloc_root进行相应的初始化。下面给出了init_alloc_root函数的源代码：

```
// mysys/my_alloc.c

1. void init_alloc_root(MEM_ROOT *mem_root, size_t block_size, size_t pre_alloc_size)
2. {
3.     mem_root->free= mem_root->used= mem_root->pre_alloc= 0;
4.     mem_root->min_malloc= 32;
5.     mem_root->block_size= block_size;
6.     mem_root->error_handler= 0;
7.     mem_root->block_num= 4;
8.     mem_root->first_block_usage= 0;

9.     if (pre_alloc_size)
10.    {
11.        if ((mem_root->free= mem_root->pre_alloc=
               (USED_MEM*)my_malloc(pre_alloc_size +
               ALIGN_SIZE(sizeof(USED_MEM)), MYF(0))))
12.        {
13.            mem_root->free->size=
                   pre_alloc_size + ALIGN_SIZE(sizeof(USED_MEM));
14.            mem_root->free->left= pre_alloc_size;
15.            mem_root->free->next= 0;
16.        }
17.    }

18.    DBUG_VOID_RETURN;
19.}

20. #define MY_ALIGN(A,L)    (((A) + (L) - 1) & ~((L) - 1))
21. #define ALIGN_SIZE(A) MY_ALIGN((A),sizeof(double))
```

先来看下init_mem_root函数的参数，mem_root指向需要进行初始化的MEM_ROOT结构。block_size是新分配内存块大小的基数。MariaDB在使用MEM_ROOT的过程中，根据不同的应用场景设置的block_size的大小也不同，从512到4096、8192、16384，等等。参数pre_alloc_size如果不为0，那么在初始化的过程中会预分配一块内存块放入空闲块链表中用于后续程序的内存申请。

在init_mem_root函数的后面，我们给出ALIGN_SIZE宏的定义，该宏的作用是进行内存对齐。

第3行到第8行代码用于初始化MEM_ROOT的成员。我们看到min_malloc的值为32，也就是说，如果USED_MEM内存块的空闲内存小于32字节，就会将其从MEM_ROOT的free链表移动到used链表。

在第9行到第17行代码中，如果参数pre_alloc_size不为0，预分配指定大小的内存块用于后续的内存申请。

4.1.5　分配内存

通常，当程序需要申请内存的时候，我们需要调用glibc的malloc函数向内存分配器申请所需的内存。如果使用MEM_ROOT来管理内存的话，就需要调用alloc_root函数向指定的MEM_ROOT申请所需的内存。下面给出了alloc_root函数的定义：

```
// mysys/my_alloc.c

1. void *alloc_root(MEM_ROOT *mem_root, size_t length)
2. {
3.     size_t get_size, block_size;
4.     uchar *point;
5.     register USED_MEM *next= 0;
6.     register USED_MEM **prev;

7.     length= ALIGN_SIZE(length);
8.     if ((*(prev= &mem_root->free)) != NULL) // 查找空闲块
9.     {
           // 如果空闲列表中的第一个块多次不能满足内存分配请求，
           // 将其移动到已用链表
10.        if ((*prev)->left < length && mem_root->first_block_usage++
               >= ALLOC_MAX_BLOCK_USAGE_BEFORE_DROP &&
               (*prev)->left < ALLOC_MAX_BLOCK_TO_DROP)
11.        {
12.            next= *prev;
13.            *prev= next->next;
14.            next->next= mem_root->used;
15.            mem_root->used= next;
16.            mem_root->first_block_usage= 0;
17.        }
18.        for (next= *prev ; next && next->left < length ; next= next->next)
19.            prev= &next->next;
20.     }
21.     if (! next) // 如果没有足够大的空闲块，则分配新的块
22.     {
23.        block_size= (mem_root->block_size & ~1) * (mem_root->block_num >> 2);
24.        get_size= length+ALIGN_SIZE(sizeof(USED_MEM));
25.        get_size= MY_MAX(get_size, block_size);
26.        if (!(next = (USED_MEM*) my_malloc(get_size,
               MYF(MY_WME | ME_FATALERROR |
```

```
                MALLOC_FLAG(mem_root-> block_size)))))
27.         {
28.             if (mem_root->error_handler)
29.                 (*mem_root->error_handler)();
30.                 DBUG_RETURN((void*) 0);
31.         }
32.         mem_root->block_num++;
33.         next->next= *prev;
34.         next->size= get_size;
35.         next->left= get_size-ALIGN_SIZE(sizeof(USED_MEM));
36.         *prev=next;
37.     }

        // 分配内存
38.     point= (uchar*) ((char*) next+ (next->size-next->left));

39.     if ((next->left-= length) < mem_root->min_malloc)
40.     {   //如果内存的剩余字节低于指定的最低值，将其移入到used链表中
41.         *prev= next->next;
42.         next->next= mem_root->used;
43.         mem_root->used= next;
44.         mem_root->first_block_usage= 0;
45.     }

46.     DBUG_RETURN((void*) point);
47.}
```

在第5行和第6行代码中，我们看到使用了register关键字的修饰变量next和prev。通常情况下，变量是存储在内存中的，经过register修饰的变量会直接存储在寄存器中，对它们的存取操作不会经过内存，所以速度很快，但同时，被register关键字修饰的变量不能做取地址运算。register通常修饰那些被频繁使用的变量。

第7行代码对请求的内存大小进行对齐。

第8行到第20行代码从空闲块链表中寻找能够满足内存分配申请的第一个空闲块。

在第10行到第17行代码中，如果空闲块链表中的第一个空闲块多次没有满足内存分配申请，说明其剩余的内存不足以满足目前情况下的内存申请，它的存在会影响内存分配的效率，所以将其移动到used链表中。

第21行到第37行代码的主要工作是当MEM_ROOT中没有能够满足当前内存申请的空闲块时，新分配一个内存块。第23行代码中，新分配内存块的大小会随着分配内存块的块数增加而呈一定关系的增大，每分配4个内存块后，接下来新分配内存块的大小在原来的基础上增加block_size设定的大小。

第38行和第39行代码用于进行内存分配，更新内存块的剩余字节数，如果内存块的剩余字节

数低于指定的阈值，则将其移动到used链表中。

最后返回分配的内存给应用程序。

从alloc_root函数的实现可以看出，MEM_ROOT向内存分配器申请大块连续的内存，应用程序向MEM_ROOT申请小块的内存，简化了内存分配器的管理工作，减少了内存分配时malloc的调用次数，同时也减少了内存碎片的产生。

4.1.6　内存回收

在使用完MEM_ROOT之后，需要调用free_root函数对申请的内存块进行回收，相关代码如下：

```
// mysys/my_alloc.c

1. static inline void mark_blocks_free(MEM_ROOT *root)
2. {
3.     register USED_MEM *next;
4.     register USED_MEM **last;

       // 将空闲块链表中的所有块设置成未使用
5.     last= &root->free;
6.     for (next= root->free; next; next= *(last= &next->next))
7.     {
8.         next->left= next->size - ALIGN_SIZE(sizeof(USED_MEM));
9.     }

       // 将空闲块链表和已用块链表进行合并
10.    *last= next=root->used;

       // 将原已用块链表中的所有内存块设置成未使用状态
11.    for (; next; next= next->next)
12.    {
13.        next->left= next->size - ALIGN_SIZE(sizeof(USED_MEM));
14.    }

15.    root->used= 0;
16.    root->first_block_usage= 0;
17.}


18.void free_root(MEM_ROOT *root, myf MyFlags)
19.{
20.    register USED_MEM *next,*old;

       // 仅仅将所有内存块标记为free，并不归还给内存分配器
21.    if (MyFlags & MY_MARK_BLOCKS_FREE)
22.    {
```

```
23.         mark_blocks_free(root);
24.         DBUG_VOID_RETURN;
25.     }

        //  MY_KEEP_PREALLOC标志表示保留预分配的内存块
26.     if (!(MyFlags & MY_KEEP_PREALLOC))
27.         root->pre_alloc=0;

        // 释放已使用链表中的所有内存块
28.     for (next=root->used; next ;)
29.     {
30.         old=next; next= next->next ;
31.         if (old != root->pre_alloc)
32.         {
33.             old->left= old->size;
34.             my_free(old);
35.         }
36.     }

        // 释放空闲链表中的所有内存块
37.     for (next=root->free ; next ;)
38.     {
39.         old=next; next= next->next;
40.         if (old != root->pre_alloc)
41.         {
42.             old->left= old->size;
43.             my_free(old);
44.         }
45.     }
46.     root->used=root->free=0;

        // 重置预分配块
47.     if (root->pre_alloc)
48.     {
49.         root->free=root->pre_alloc;
50.         root->free->left=root->pre_alloc->size-ALIGN_SIZE(sizeof(USED_MEM));
51.         root->free->next=0;
52.     }

53.     root->block_num= 4;
54.     root->first_block_usage= 0;
55.     DBUG_VOID_RETURN;
56.}
```

下面我们先来看mark_blocks_free函数，该函数的主要功能是将MEM_ROOT中的所有内存块设置成空闲状态。

第5行到第9行代码用于将free链表中的所有内存块设置成未使用状态。

第10行代码用于将used链表合并到空闲链表中。

第11行到第14行代码用于将原used链表中的所有内存块设置成未使用状态。

接下来我们分析free_root函数。该函数有两个参数，第一个参数root指向需要释放的MEM_ROOT结构；第二个参数MyFlags是标志位，可以控制free_root的行为。如果MyFlags携带了MY_MARK_BLOCKS_FREE标志，表明仅仅将MEM_ROOT中的所有内存块设置成空闲状态，而不归还给内存分配器，以便后续重复使用；如果MyFlags携带了MY_KEEP_PREALLOC标志，将不会释放预分配的内存块。

第21行到第25行代码中，如果使用了MY_MARK_BLOCKS_FREE标志，仅仅将所有内存块设置成空闲状态，然后直接返回。

第28行到第36行代码用于释放free链表中的所有内存块。

第37行到第45行代码用于释放used链表中的所有内存块。

4.1.7　MEM_ROOT 的使用场景

通过以上的介绍，我们相信读者对MEM_ROOT的使用及其内存管理机制有了一个很深刻的理解。细心的读者会发现用户通过标准库的malloc函数申请的内存在使用完之后需要调用free函数将其释放，而用户向MEM_ROOT申请的内存不需要执行相应的free操作，只需要在最后将MEM_ROOT中的所有内存块置为空闲或者释放掉即可，这样的好处是管理起来十分简单，不需要跟踪每个已分配的内存在什么时候进行回收，缺点就是这样的策略限制了MEM_ROOT的使用场景——使用同一个MEM_ROOT实例的所有对象必须具有相同的生命周期。所以我们在使用MEM_ROOT的时候遵循了一个这样的原则：具有相同生命周期的变量向同一个MEM_ROOT实例申请内存，在这些变量的生命周期结束之后将MEM_ROOT中的内存释放就可以了，具有不同生命周期的变量向不同的MEM_ROOT实例申请内存。

4.2　文件缓存 IO_CACHE

缓存技术在计算机领域中被广泛用于缓存结果，减少慢速操作的执行次数，从而提高系统的整体性能。例如，浏览器缓存缓存了网站的数据，减少了对网站的访问，提高了用户体验；许多门户网站将主站的数据缓存到分布在各地的CDN结点上，用户的读取请求会被分发到距离最近的CDN结点上，从而提高请求的响应速度。

磁盘I/O是一个相对来说比较慢速的操作，MariaDB通过IO_CACHE结构缓存文件的读写操作，减少了磁盘的I/O次数，提高了磁盘文件的读写效率。

4.2.1　高性能武器——缓存

缓存，作为计算机系统中非常重要的一部分，存在于计算机系统中的许多位置，对提高系统的整体性能发挥着举足轻重的作用。同时，缓存技术对于构建高性能的服务架构也非常有用。

1. CPU L1/L2 cache

在计算机的发展过程中，CPU的运行速度越来越快，CPU要求从内存中读取数据的速度也越来越快。然而内存的速度提升却很缓慢，同时能够高速读写的内存价格又非常昂贵，不能大量地采用。著名的CPU生产厂商Intel从性价比的角度出发，采用少量的高速内存和大量的廉价内存相结合的方式，减少CPU访问内存的平均时间，这就是CPU cache技术。CPU cache就是少量的位于CPU和主存之间的能够高速读写的内存。最初的CPU cache只有一级（CPU L1 cache），随着CPU速度的进一步提升，慢慢出现了二级缓存（CPU L2 cache），它位于一级缓存和主存之间，二级缓存的容量比一级缓存大，但速度相对于一级缓存要慢一些，但比内存要快得多。现在许多主流的CPU已经包含了三级缓存（CPU L3 cache）。图4-4演示了主存、一级缓存和二级缓存之间的关系。

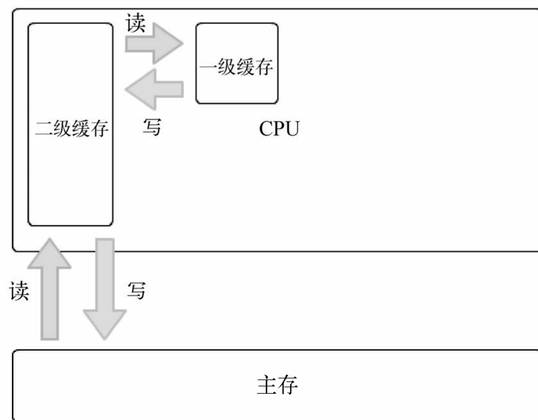


图4-4　CPU L1/L2 cache

2. 页高速缓存

在计算机系统中，访问一次内存所需要的时间在20ns ~ 100ns（纳秒），而访问一次磁盘所需的时间在10ms（毫秒）左右（包括寻道时间+旋转延时+数据传输时间），两者之间相差数10万倍。Linux内核采用page cache机制（页面高速缓存，页面的大小为4KB）加快对磁盘文件的读写。页高速缓冲区是内存中用于缓存磁盘文件数据的一块区域。对于磁盘文件的读请求，内核首先在页高速缓冲区中查找，如果存在，直接从页高速缓冲区中读取所请求的数据，如果页高速缓冲区中没有，需要从磁盘读取对应的页，同时读取紧随其后的少数几个页，并将其放入页高速缓冲区中来；对于磁盘文件的写请求，内核直接修改页高速缓冲区中对应的页，被修改过的页称为脏页，

内核负责定期地（或者由于脏页达到一定比率）将页高速缓冲区中的脏页刷新到磁盘中，这种写策略也叫作回写（write back），如图4-5所示。

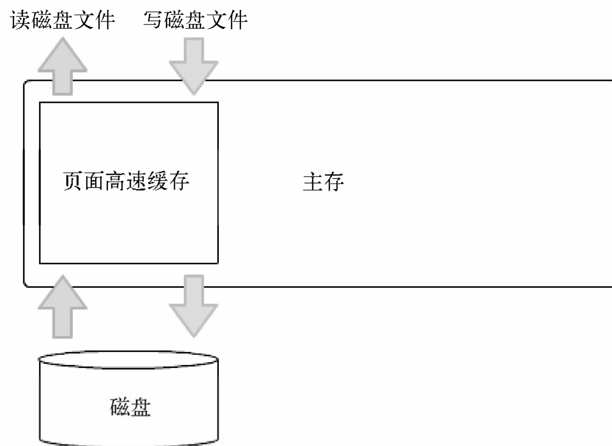


图4-5　页面高速缓存

3. 分布式内存缓存——memcached

memcached是一个高性能的分布式内存缓存。在一些网站架构中，它经常被用来缓存一些数据库查询结果或者是需要计算的中间结果，减少慢速的数据库访问操作和计算工作，从而提高整个网站的响应速度以及网站的负载能力。

缓存，无论位于系统层，还是应用层，位于分布式架构中，还是单机系统里，其本质都是减少慢速操作的次数，提高系统的整体性能。

4.2.2　IO_CACHE 的定义

IO_CACHE是MariaDB中定义的I/O缓存结构，主要作用是减少磁盘操作的次数，从而提高磁盘文件的读写性能。除此之外，IO_CACHE还可以被用来作为网络IO的缓冲区。在MariaDB中，许多磁盘文件的操作都是通过IO_CACHE进行的。例如所有的日志文件的读写，包括二进制日志binlog的读写、查询日志general log的读写，等等。

下面我们列举了枚举类型cache_type以及IO_CACHE结构本身的定义：

```
// include/my_sys.h

enum cache_type
{
    TYPE_NOT_SET= 0,
    READ_CACHE,
```

```
    WRITE_CACHE,
    SEQ_READ_APPEND                     /* 顺序读取追加写入 */,
    READ_FIFO,
    READ_NET,WRITE_NET
};

typedef struct st_io_cache
{
    enum cache_type type;
    int file;                           // 文件描述符
    uchar  *buffer;
    size_t buffer_length;
    my_off_t pos_in_file;               // buffer中的数据在文件中的偏移量
    my_off_t end_of_file;               // 文件结尾位置
    uchar *read_pos;
    uchar *read_end;
    uchar *request_pos;
    uchar *write_buffer;
    uchar *write_pos;
    uchar *write_end;
    uchar *append_read_pos;
    uchar **current_pos, **current_end;
    mysql_mutex_t append_buffer_lock;
    IO_CACHE_SHARE *share;
    int (*read_function)(struct st_io_cache *,uchar *,size_t);
    int (*write_function)(struct st_io_cache *,const uchar *,size_t);

    // 可以注册相关的回调函数，在不同的阶段执行
    IO_CACHE_CALLBACK pre_read;
    IO_CACHE_CALLBACK post_read;
    IO_CACHE_CALLBACK pre_close;
    void *arg;
    ulong disk_writes;                  // 统计写磁盘次数，也就是将写缓冲区的内容写入到文件中的次数
    int seek_not_done;
    int error;
    size_t  read_length;
    myf myflags;
    my_bool allocated_buffer;
} IO_CACHE;
```

从cache_type的定义可以看出，IO_CACHE可以作为磁盘文件的读缓存、写缓存或者是顺序读取追加写入缓存，也可以作为网络IO的缓存。

下面我们给出IO_CACHE各个成员的含义。

- type：IO_CACHE的类型，可以是READ_CACHE或者是WRITE_CACHE，等等。
- file：与IO_CACHE关联的文件描述符。
- buffer：IO_CACHE的缓冲区，一般情况下是在IO_CACHE初始化阶段分配的。

- buffer_length：缓冲区的大小。
- pos_in_file：缓冲区的内容在文件中的偏移量。
- end_of_file：文件的结束位置。
- read_pos：已读取的数据在读缓存中的偏移量。
- read_end：读缓存中可以读取的数据的结束位置。
- request_pos：当使用的是async_io的时候使用。
- write_buffer：写缓冲区的开始位置。当IO_CACHE的类型为WRITE_CACHE时，write_buffer的值和buffer的值一致；当IO_CACHE的类型为SEQ_READ_APPEND时，write_buffer指向buffer的正中间位置，因为在这种情况下，缓冲区buffer的前面一半部分作为顺序读缓冲区，后面的另一半作为追加写缓冲区。
- write_pos：下一次写应该从write_pos指定的位置开始。
- write_end：写缓冲区的末尾。
- append_read_pos：当IO_CACHE的类型为SEQ_READ_APPEND时，当顺序读缓冲区没有足够的数据满足用户的读请求时，会去文件中读取相应的数据，如果文件中也没有足够的数据时，会去追加写缓冲区读取相应的数据。append_read_pos指示了已经读取到了追加写缓冲区的相应位置。
- append_buffer_lock：上面介绍过，当IO_CACHE的类型为SEQ_READ_APPEND时，可能会读取追加写缓冲区中的内容，为了防止在读取追加写缓冲区的过程中发生写操作，需要进行相应的锁保护。
- share：主要用在有多个线程对同一个文件进行并发读时。
- read_function和write_function：它们为两个函数指针，其所对应的函数根据IO_CACHE的类型而有所不同。

  - read_function函数的主要作用是当读缓冲区中的内容读完之后，将文件中的内容（或者是append_buffer中的内容）读取到读缓存中。
  - write_function函数的主要作用是当写缓冲区没有足够的空间来满足当前的写操作时，将写缓冲区的内容写入到文件中。

- pre_read、post_read和pre_close：这是3个回调函数，如果设置了相应的值，将会在执行IO操作的某些阶段执行用户指定的动作。如果指定了pre_read的值，在IO_CACHE发生实际的读操作（从文件读取数据到读缓冲区）之前会执行pre_read指定的操作；如果post_read的值不为空，在IO_CACHE发生实际的读操作之后会执行post_read指定的操作；如果指定了pre_close的值，在结束IO_CACHE前会执行pre_close指定的操作。
- arg：pre_read和post_read回调函数的参数。
- disk_writes：写磁盘操作的次数，也就是将写缓冲区的内容写入到文件中的次数。
- seek_not_done：如果为true，表明没有执行相应的seek操作，主要用于从指定位置读或写文件。

- error：错误码。
- read_length：read_length的值通常情况下和buffer_length的值一致，除非当前使用的是async_io。
- myflages：主要指定在执行真正的读写操作时，当发生了错误，需要进行什么样的操作，例如是否输出错误信息，等等。
- allocated_buffer：如果为true，说明当前的缓冲区是在初始化IO_CACHE的过程中分配的，在结束IO_CACHE的时候需要对其进行释放。

4.2.3 IO_CACHE 的使用

本节中我们介绍IO_CACHE作为几种不同类型缓存的使用，这几种类型为磁盘文件读缓存、磁盘文件写缓存以及磁盘文件顺序读取追加写入缓存。

4

1. IO_CACHE作为磁盘文件读缓存

我们知道，从磁盘文件中读取数据是比较耗时的操作，所以在Linux系统中采用了页缓存机制来提高磁盘文件的读写效率。页缓存机制是位于操作系统层面的磁盘文件缓存，我们也可以在应用层面为磁盘文件添加缓存，从而提高磁盘文件的读写效率。IO_CACHE可以作为磁盘文件位于应用层面的读缓存，用来减少磁盘文件的读操作，提高磁盘文件的读性能。

磁盘文件读缓存的一般策略如下：读操作首先查看缓存中是否包含所请求的数据，如果有，直接从缓存中读取，如果没有，从磁盘文件读取包含所请求数据的一大块数据放入缓存中。磁盘文件的读缓存之所以能够提高磁盘文件的读效率，主要归功于数据的局部性原理，也就是说下次读操作所请求的数据很可能位于这一次读请求所读取数据的附近。

下面给出了一个IO_CACHE作为读缓存的例子：

```
/*=============example 4-3：IO_CACHE作为读缓存=============*/

// 初始化缓存
IO_CACHE file_cache
if (init_io_cache(&file_cache, fd, cache_size, READ_CACHE, start_pos, 0, 0))
{
    printf("init_io_cache failed\n");
    return init_error;
}
...

// 读取指定字节的数据
char buffer[BUFFER_LEN];
my_b_read(&file_cache, buffer, BUFFER_LEN);
...
```

```
// 读取一个字节
uchar one_byte = my_b_get(&file_cache);
...

// 查询读取到文件的哪个位置
size_t positon = my_b_tell(&file_cache);
...

// 结束IO_CACHE，释放分配的内存
if (end_io_cache(&file_cache))
{
    printf("end_io_cache failed.\n");
    return end_error;
}
```

在使用IO_CACHE之前必须调用init_io_cache对其进行初始化操作。init_io_cache函数的声明如下：

```
// include/my_sys.h文件

int init_io_cache(IO_CACHE *info, int file, size_t cachesize,
    enum cache_type type, my_off_t seek_offset,
    pbool use_async_io, myf cache_myflags);
```

下面我们介绍下init_io_cache各个参数的具体含义。

- 参数info指向需要初始化的IO_CACHE结构，该参数不能为空。
- 参数file为IO_CACHE相关联文件的描述符。
- 参数cachesize将指定分配多大的内存作为IO_CACHE的缓存。
- 参数type指定了IO_CACHE的作用，当IO_CACHE作为读缓存的时候该参数为READ_CACHE，当IO_CACHE作为写缓存的时候该参数为WRITE_CACHE。
- 参数seek_offset指定了缓存中的内容在文件中的偏移量。如果参数use_async_io指定为1，同时系统支持async_io（异步IO），将使用异步IO接口进行读写操作。
- 用户可以通过参数cache_myflags指定init_io_cache的动作，例如是否检查文件的大小，等等。

当IO_CACHE作为读缓存的时候，init_io_cache函数主要完成了以下工作。

- 将传入的文件描述符和当前IO_CACHE进行关联。
- 如果传入的seek_offset不为0，移动到文件的指定位置。
- 分配内存作为缓存区。

在对IO_CACHE进行初始化之后，你就可以对IO_CACHE进行读取操作了。MariaDB定义了两个接口对IO_CACHE进行读取操作，其声明如下：

```
// include/my_sys.h文件

void my_b_read(IO_CACHE *info, uchar *buffer, size_t count);

int my_b_get(IO_CACHE *info);
```

my_b_read函数的作用是从IO_CACHE读取指定字节的数据到buffer中。而my_b_get的作用是从IO_CACHE读取一个字节。

当你想知道读取到文件中的哪个位置时，可以调用my_b_tell函数来查询相关的位置信息：

```
// include/my_sys.h文件

size_t my_b_tell(IO_CACHE *info);
```

在使用完IO_CACHE之后，需要调用end_io_cache来完成相应的清理工作。在IO_CACHE作为读缓存的时候，end_io_cache的主要工作是释放已分配的内存，以免发生内存泄漏：

```
// include/my_sys.h文件

int end_io_cache(IO_CACHE *info);
```

图4-6给出了IO_CACHE作为读缓存的工作机制。用户调用my_b_get函数和my_b_read函数从IO_CACHE中读取所需的数据。当IO_CACHE的缓冲区中的内容被全部读取完之后，IO_CACHE会使用my_b_read函数从文件的相应位置读取一整块数据到IO_CACHE的缓冲区。对于用户来说，可见的部分只有方框外的my_b_get函数和my_b_read函数。

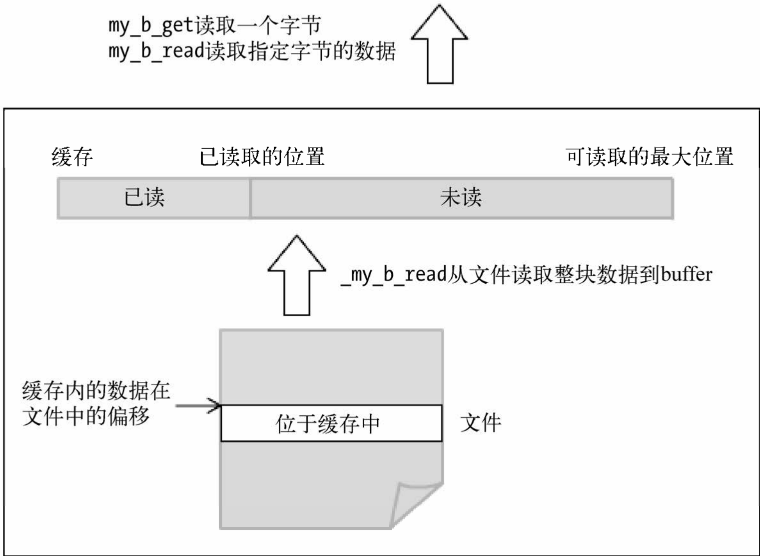


图4-6 IO_CACHE作为读缓存的工作机制

2. IO_CACHE作为写缓存

当然，IO_CACHE也可以作为磁盘文件的写缓存，减少对磁盘文件写的次数，提高磁盘文件的写性能。

下面给出了一个使用IO_CACHE作为写缓存的例子：

```
/*=============example 4-4：IO_CACHE作为写缓存=============*/

// 初始化缓存
IO_CACHE file_cache
if (init_io_cache(&file_cache, fd, cache_size, WRITE_CACHE, start_pos, 0, 0))
{
    printf("init_io_cache failed\n");
    return init_error;
}
...

// 写一块数据到IO_CACHE
uchar buffer[buffer_len];
...
my_b_write(&file_cache, buffer, buffer_len);

// 写一个字节到IO_CACHE
uchar chr = 'a';
my_b_write_byte(&file_cache, chr);
...

// 查询写文件的哪个位置
size_t positon = my_b_wirte_tell(&file_cache);
...

// 结束IO_CACHE，将缓存中的内容刷新到文件中，并且释放分配的内存
if (end_io_cache(&file_cache))
{
    printf("end_io_cache failed.\n");
    return end_error;
}
```

和读缓存一样，IO_CACHE作为写缓存，在使用之前也需要调用init_io_cache函数进行初始化，不同的是，此时type参数的值为WRITE_CACHE。

接下来就可以调用my_b_write函数或my_b_write_byte函数进行写操作。这两个函数的声明如下：

```
// include/my_sys.h文件

void my_b_write(IO_CACHE *info, uchar *buffer, size_t count);

void my_b_write_byte(IO_CACHE *info, uchar chr);
```

my_b_write函数的功能是往IO_CACHE中写入一块数据，而my_b_write_byte函数的功能是往IO_CACHE中写入一个字节。

你还可以通过my_b_write_tell函数来查询当前写的数据在文件中的偏移量。my_b_write_tell函数的声明如下：

```
// include/my_sys.h文件

size_t my_b_write_tell(IO_CACHE *info);
```

当用完IO_CACHE时，需要调用end_io_cache进行相应的清理工作。当IO_CACHE作为写缓存的时候，end_io_cache的主要功能如下。

- 将IO_CACHE缓存中的内容刷入到文件。
- 释放缓冲区分配的内存。

图4-7给出了IO_CACHE作为写缓存的工作机制，用户调用my_b_write函数和my_b_write_byte函数进行写操作。当写缓冲区的数据达到一定量之后，没有更多的空间用于满足当前的写请求时，IO_CACHE会调用_my_b_write函数将缓存中的内容写入到文件中。

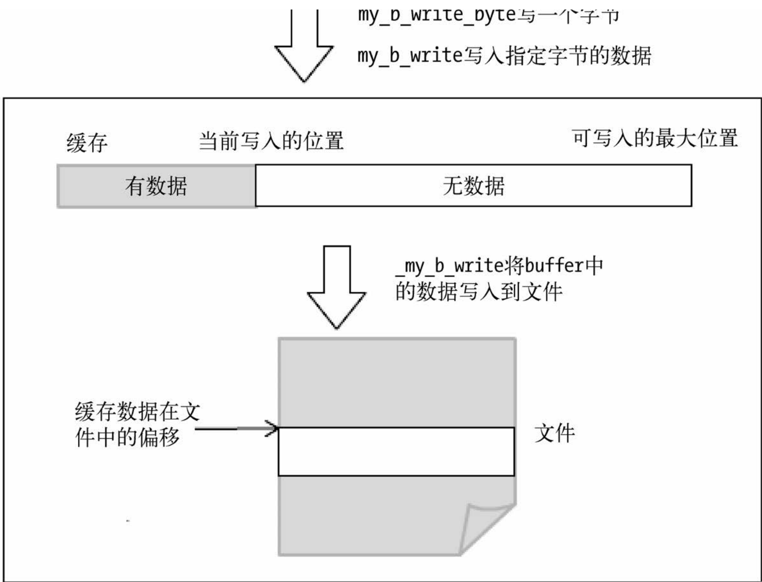


图4-7 IO_CACHE作为写缓存

3. IO_CACHE作为顺序读取追加写入缓存

除了上面介绍的可以作为磁盘文件的读缓存和写缓存外，IO_CACHE还可以同时作为文件的顺序读缓存和追加写缓存。顺序读表示只能从前往后读取，追加写表示只能在文件的末尾追加数据。

首先我们给出一个使用IO_CACHE作为顺序读取追加写入缓存的例子：

```
/*============example 4-5：IO_CACHE作为顺序读取追加写入缓存============*/

// 初始化缓存
IO_CACHE file_cache;
if (init_io_cache(&file_cache, fd, cache_size, SEQ_READ_APPEND, start_pos, 0, 0))
{
    printf("init_io_cache failed\n");
    return init_error;
}
...

// 从IO_CACHE顺序读取数据
uchar read_buffer[read_len];
...
my_b_read(&file_cache, read_buffer, read_len);
uchar chr = my_b_get(&file_cache);
...

// 追加数据到文件末尾
uchar append_buffer[append_len];
...
if (error = m_b_append(&file_cache, append_buffer, append_len))
{
    printf("append file_cache failed.\n");
    return error;
}

// 结束IO_CACHE
if (end_io_cache(&file_cache))
{
    printf("end_io_cache failed.\n");
    return end_error;
}
```

当IO_CACHE作为顺序读取追加写入缓存时，读取数据的接口是不变的，还是使用my_b_read函数和my_b_get函数，但写数据的接口发生了变化，使用的是my_b_append函数，该函数负责将数据追加到文件末尾，其声明如下：

```
// include/my_sys.h

int my_b_append(IO_CACHE *info, uchar *append_buffer, size_t append_len);
```

end_io_cache函数在IO_CACHE为顺序读取追加写入缓存时的主要工作和IO_CACHE作为写缓存的功能基本一致，一方面将append_buffer中的数据刷入到文件中，另一方面释放已分配的内存。

图4-8给出了IO_CACHE作为顺序读取追加写入缓存的工作机制。我们需要注意的是，此时分配

了两倍于cachesize大小的内存，前半部分作为顺序读缓冲区（seq_read_buffer），而后半部分作为追加写缓冲区（append_buffer）。

当从顺序读取追加写入类型的IO_CACHE中读取数据时，读取数据的优先级如下。

(1) 首先从seq_read_buffer中读取。
(2) 其次从文件中读取。
(3) 最后从append_buffer中读取。

当seq_read_buffer中有足够的数据可以满足读请求时，直接从中读取相应的数据即可。当seq_read_buffer中没有足够的数据时，会去文件中读取相应的数据。如果文件中也没有足够的数据，那么就从append_buffer中读取相应的数据。在读取append_buffer中的内容时需要进行加锁操作，因为其他的进程可能会对append_buffer进行写操作。

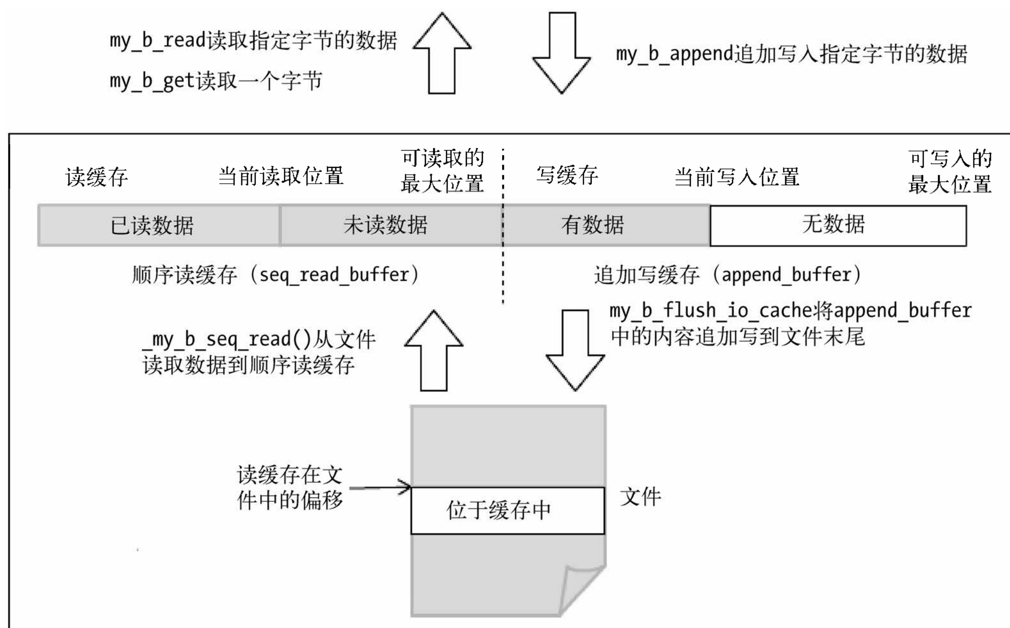


图4-8　IO_CACHE作为顺序读取追加写入缓存

4.3　NET 结构

NET结构定义了所有网络相关的操作。在MariaDB中，NET主要用于客户端和服务端之间的通信，同时，NET内部还支持对数据进行压缩。图4-9描述了NET结构以及对NET进行操作的相关函数。Vio结构定义了底层网络I/O的相关操作。在NET内部，包含一个缓冲区，服务端从客户端接收到的数据会放入缓冲区，服务端向客户端发送数据时也会用到该缓冲区。

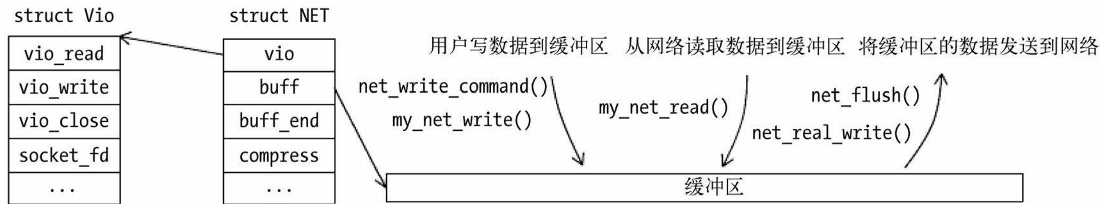


图4-9　NET

接下来，我们给出NET结构的定义：

```
// include/mysql_com.h

typedef struct st_net {
    Vio *vio;
    unsigned char *buff;
    unsigned char *buff_end;
    unsigned char *write_pos;
    unsigned char *read_pos;

    my_socket fd;

    unsigned long remain_in_buf,length, buf_length, where_b;
    unsigned long max_packet,max_packet_size;

    unsigned int write_timeout, read_timeout, retry_count;
    unsigned int *return_status;
    unsigned char reading_or_writing;
    char save_char;

    my_bool compress;                    // 如果为true，表示和客户端之间通信时数据进行压缩

    unsigned int pkt_nr;                 // 当前包序号
    unsigned int compress_pkt_nr;        // 压缩协议的当前包序号

    unsigned int last_errno;             // 发送给客户端的最后一个错误码（MySQL内部错误码）
    unsigned char error;

    char last_error[MYSQL_ERRMSG_SIZE];

    char sqlstate[SQLSTATE_LENGTH+1];
} NET;
```

下面我们给出NET的各个成员的具体含义。

- vio：vio封装了底层网络I/O的相关操作。
- buff、buff_endbuff和buff_end：它们定义了缓冲区，用于缓存读取的客户端数据或者将要发送到客户端的数据。buff是缓冲区的开始，buff_end是缓冲区的结束。

- write_pos：当缓冲区缓存了将要发送到客户端的数据时，write_pos指向下一次写缓冲区的开始位置。
- read_pos：当缓冲区缓存了客户端发来的数据时，read_pos指向下一次读取数据的开始位置。
- fd：用于支持Perl DBI/DBD客户端接口。
- remain_in_buff：在压缩协议中，在读取压缩包的过程中，可能会尝试读取比压缩包长度更长的网络数据，因此读取的数据可能包括下一个包的部分数据，该变量用于记录读取了多少字节下一个包的数据。在没有使用压缩协议进行通信时，不会用到该变量。
- length：当前包的长度，不包含包头的长度。
- buf_length：缓冲区中有效的数据长度。
- where_b：read_pos-buff的值，缓冲区中当前读取的位置。
- max_packet：当前缓冲区的长度。
- max_packet_size：允许的包的最大值，对应于配置文件中的max-allowed-packet。
- read_timeout和write_timeout：网络读操作的超时时间和网络写操作的超时时间，分别对应于配置文件中的net_read_time和net_write_timeout。
- return_status：指向与该连接相关联的THD线程描述符中的server_status变量。
- retry_count：网络操作失败时重试的次数,对应于配置文件中的net_retry_count。
- reading_or_writing：在没有正在进行的I/O操作时为0，有读操作时为1，有写操作时为2。
- compress：如果为1，表明与客户端通信时对数据进行压缩。
- pkt_nr：当前包序号。
- compress_pkt_nr：使用压缩协议进行通信时的当前包序号。
- last_errno：发送给客户端的最后一个错误信息的错误码。
- error：如果I/O操作成功，则设置成0；如果协议层上有逻辑错误，则设置成1；如果有系统调用或标准库错误，则设置成2；特殊情况下，在尝试展开一个缓冲区，接受一个大型包失败后，如果成功跳过这个大型包，则设置成3。
- sqlstate：SQL的状态。

4.4 线程上下文——THD

THD类是一个线程描述符，包含了一个线程的所有上下文信息。通常情况下，一个THD实例对应于一个线程，但在某些情况下，例如MariaDB使用的是线程池模式来处理客户端连接的话，每个客户端的连接也都会对应一个THD实例。

如果THD实例对应的是一个客户端的连接，那么它包含了当前连接的所有信息，例如客户端的IP地址信息、客户端登录所使用的账号、当前用户的权限信息、当前选择的是哪个数据库（USE DB命令）、当前执行的事务状态、当前连接上执行的查询是否被杀死了，等等。THD实例还可能对应于MariaDB中的其他系统线程，例如用于复制的slave线程，等等。

下面我们给出类THD的定义，其中仅仅列出一些比较重要的成员：

```
// sql/sql_class.h

class THD {
public:
    LEX *lex;                          // 当前查询语法树
    CSET_STRING query_string;
    char *db;                          // 当前选择的数据库
    size_t db_length;

    TABLE *open_tables;
    TABLE *temporary_tables;
    TABLE *derived_tables;

    MYSQL_LOCK *lock;

    MDL_context mdl_context;           // 元数据锁
    NET net;

    Protocol *protocol;
    HASH    user_vars;                 // 用户自定义的变量
    String  packet;                    // 网络I/O动态缓冲区
    String  convert_buffer;

    struct  system_variables variables;
    mysql_mutex_t LOCK_thd_data;

    char *thread_stack;                // 当前线程栈起始地址
    const char *where;

    ulong client_capabilities;         // 客户端支持的功能
    ulong max_client_packet_length;

    enum enum_server_command m_command;
    uint16 peer_port;                  // 对端端口号

    struct {
        bool       report_to_client;
        bool       report;
        uint       stage, max_stage;
        ulonglong  counter, max_counter;
        ulonglong  next_report_time;
        Query_arena *arena;
    } progress;                        // 当前命令的执行进度

private:
    binlog_filter_state m_binlog_filter_state;
    enum_binlog_format current_stmt_binlog_format;
```

```
public:
    struct st_transactions transaction;

    Global_read_lock global_read_lock;
    table_map table_map_for_update;

private:
    ha_rows    m_sent_row_count;
    ha_rows    m_examined_row_count;

public:
    USER_CONN *user_connect;
    CHARSET_INFO *db_charset;

    query_id_t query_id;
    pthread_t  real_id;                // POSIX线程号
    my_thread_id  thread_id;

    enum_tx_isolation tx_isolation;      // 当前连接的事务隔离级别

    killed_state volatile killed;        // 终止标识

    bool slave_thread;                 // 如果为1，表明当前线程为slave_io线程或者slave_sql线程
    NET *slave_net;                    // 从库到主库的网络连接

    time_t current_connect_time;

    Locked_tables_list locked_tables_list;

    MEM_ROOT main_mem_root;            // 内存池
    ...
};
```

下面我们给出THD中各个成员的含义。

- lex：当前查询的语法解析树。
- query_string：当前查询的字符串形式。
- db和db_length：当前选择的数据库，可以通过use命令来更改。
- open_tables：当前会话打开的所有普通表，包括打开的数据库表和临时表。
- temporary_tables：当前会话打开的所有临时表，包括使用create temporary table命令打开的用户层面的临时表以及某些命令在执行过程中使用到的内部临时表，例如alter命令。
- derived_tables：当前查询生成的派生表，例如SELECT语句的WHERE条件中如果包含子查询语句，那么会生成派生表来保存子查询的结果。
- lock：执行SELECT、INSERT或UPDATE等语句的过程中自动获取的表锁（不是通过LOCK TABLES命令获取的）。

- mdl_context：当前事务所获取的元数据锁（meta data lock）。元数据锁是相对于数据锁（data lock）来说的，其作用是保护库表结构。当执行DDL语句的时候，会首先获取表的排他元数据锁，而在执行DML语句的时候会获取共享元数据锁。
- net：到客户端的网络连接。
- protocol：客户端和服务端的通信协议描述符。
- user_vars：用户自定义的变量。用户可以在会话中自定义变量，例如下面的语句：

```
SET @a:=3;
SELECT col1 FROM t1 WHERE clo2=@a;
```

- packet：网络I/O的动态缓冲区。
- convert_buffer：编码转换的动态缓冲区。
- variables：可以被客户端改变的系统变量在当前连接中的值。例如，用户执行如下命令：

```
SET LOCAL sort_buffer_size=256000;
```

那么当前连接中执行ORDER BY和GROUP BY命令时可使用的排序缓冲区的最大值为256000。variables变量还包括server_id的值。

- LOCK_thd_data：保护当前THD数据的锁。
- thread_stack：当前线程的线程栈的开始地址，主要用于防止栈溢出。
- where：用于进行错误提示，指示客户端的哪块语句有错误。
- client_capabilities：指示客户端支持哪些功能。通常，新版本的客户端能够支持更多的通信协议。通过该变量，服务端知道选择一种客户端支持的协议与其进行通信。
- max_client_packet_length：客户端包的最大长度。
- m_command：当前查询的命令类型，例如是COM_INIT_DB或者COM_QUERY，等等。
- peer_port：该连接客户端的端口号。
- progress：该变量记录了当前命令的执行进度信息。
- m_binlog_filter_state：指示了当前语句是否该写入到binlog中。
- current_stmt_binlog_format：指示了当前语句记录binlog的格式。
- transaction：当前事务上下文。
- global_read_lock：全局读锁。
- table_map_for_update：对于某些语句可能会修改多张表的数据，该变量记录了被修改的这些表。
- m_sent_row_count：实际发送给客户端的数据行数。
- m_examined_row_count：命令执行过程中访问的数据行数，主要用于慢查询日志报告。
- user_connect：该变量是针对账号进行的一些统计信息，例如该账号同时登录的客户端个数、该账号每小时发送的写请求数，等等。
- db_charset：当前数据库的编码。

- query_id：当前查询在MariaDB内部的id号。
- real_id：当前线程的POSIX线程号，主要用于调试。
- thread_id：当前线程在MariaDB中的线程号，是我们执行SHOW PROCESSLIST时看到的线程号，可以作为KILL命令的参数。
- tx_isolation：当前连接的事务隔离级别。事务的隔离级别分为read uncommitted（可读取未提交事务）、read committed（可读取已提交事务）、repeatable read（可重复读取）和serializable（序列化）。
- killed：如果某个线程被要求停止，该变量会被设置成1，例如我们在执行KILL命令结束某个查询的时候，将对应线程的killed变量置成1。每个线程在运行的过程中有义务经常去检查该变量的值，如果发现为1，尽快进行清理工作然后退出。
- slave_thread：如果为1，表明当前线程是一个slave IO线程或者slave SQL线程。
- slave_net：如果当前线程是slave IO线程，会建立一个从库到主库的网络连接，该变量包含了该网络连接的所有信息。
- current_connect_time：当前客户端连接到MariaDB的时间。
- locked_tables_list：使用LOCK TABLES命令锁定的表。
- main_mem_root：当前线程的内存池。

4.5 TABLE_SHARE

类TABLE_SHARE定义了数据库表的基本信息，一个TABLE_SHARE实例对应于数据库中的一个表。当我们执行一条查询语句时，必须打开对应的表，这时会创建一个TABLE的实例，如果TABLE实例对应的TABLE_SHARE实例不存在（可能是第一次使用该表，或者之前执行了FLUSH TABLES命令），那么需要从.frm文件中读取该表的信息，然后创建对应的TABLE_SHARE实例。

图4-10给出了数据库表、TABLE_SHARE实例以及TABLE实例之间的关系。可以看到，一个数据库表对应一个TABLE_SHARE实例，但可以对应多个TABLE实例，一个数据库表的多个TABLE实例之间共享一个TABLE_SHARE实例。

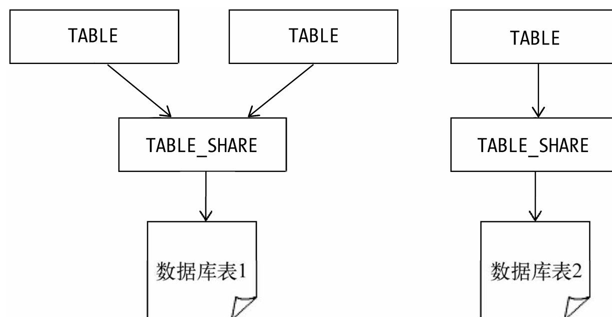


图4-10 TABLE_SHARE对象和TABLE对象的关系

下面我们给出TABLE_SHARE的定义：

```
// sql/table.h

struct TABLE_SHARE
{
    HASH name_hash;                        // 通过列名查找对应的列
    MEM_ROOT mem_root;                     // 内存池
    TYPELIB keynames;                      // 索引名索引类型
    TYPELIB fieldnames;                    // 列名列类型

    mysql_mutex_t LOCK_ha_data;
    mysql_mutex_t LOCK_share;

    typedef I_P_List <TABLE, TABLE_share> TABLE_list;
    struct
    {
        mysql_mutex_t LOCK_table_share;
        TABLE_SHARE *next, **prev;         // 未使用的TABLE_SHARE对象
        uint ref_count;                    // 使用该TABLE_SHARE的TABLE对象的个数

        Wait_for_flush_list m_flush_tickets;// 等待该TABLE_SHARE对象刷新的线程

        TABLE_list used_tables;            // 正在使用的TABLE对象
        TABLE_list free_tables;            // 未使用的TABLE对象
    } tdc;

    LEX_CUSTRING tabledef_version;         // 版本信息

    engine_option_value *option_list;
    ha_table_option_struct *option_struct;

    // 以下几项内容会复制到每个打开的TABLE对象中
    Field **field;                         // 所有的列
    KEY *key_info;                         // 该表的索引
    uint *blob_field;                      // blob类型的列在field中的位置

    uchar *default_values;                 // 列的默认值
    LEX_STRING comment;                    // 表的注释
    CHARSET_INFO *table_charset;           // 字符列的默认编码

    LEX_STRING table_cache_key;            // db + table作为查询表的键值
    LEX_STRING db;                         // 数据库名
    LEX_STRING table_name;                 // 表名
    LEX_STRING path;                       // .frm文件的路径
    LEX_STRING normalized_path;

    ha_rows min_rows, max_rows;
    ulong avg_row_length;
```

```
    ulong version;                      // 版本信息
    ulong mysql_version;                // MariaDB版本信息，对于5.0之前的版本，该值为0

    ulong stored_rec_length;            // 记录长度

    plugin_ref db_plugin;               // 存储引擎插件
    enum row_type row_type;             // 行的存储格式，包括fixed、dynamic、compact，等等
    enum tmp_table_type tmp_table;      // 临时表类型，包括事务类型的、非事务类型的、内部表，等等

    enum ha_choice transactional;       // 是否是事务类型的表
    enum ha_choice page_checksum;       // 是否开启了页校验

    uint key_block_size;
    uint stats_sample_pages;

    uint fields;                        // 列的个数，包含虚拟列
    uint stored_fields;                 // 列的个数，不包含虚拟列
    uint rec_buff_length;               // 行缓存大小

    uint null_fields;                   // 可以为NULL的列的个数
    uint blob_fields;                   // blob类型的列的个数
    uint varchar_fields;                // 字符串列的个数
    uint db_create_options;             // 创建表时的选项，例如引擎类型是否开启check_sum，等等
    uint db_options_in_use;             // 正在使用的选项
    uint db_record_offset;              // 记录在表中的开始位置
    uint rowid_field_offset;            // rowid列的位置

    uint primary_key;
    enum open_frm_error error;
    uint open_errno;
    uint vfields;                       // 虚拟列的个数
    uint default_fields;                // 有默认值列的个数
    bool crypted;                       // 如果为true，说明.frm文件是加密的
    bool crashed;                       // 崩溃标记
    bool is_view;                       // 该表的类型为视图
    bool deleting;                      // 如果为true，表明该表正在被删除
    bool can_cmp_whole_record;
    ulong table_map_id;                 // 表的内部id，用于行格式的复制

    char *tablespace;                   // 该表使用的表空间名称

// 分区表信息
#ifdef WITH_PARTITION_STORAGE_ENGINE
    bool auto_partitioned;
    char *partition_info_str;
    uint  partition_info_str_len;
    uint  partition_info_buffer_size;
    plugin_ref default_part_plugin;
#endif
};
```

下面给出部分成员的含义。

- name_hash：如果表的列数大于等于MAX_FIELDS_BEFORE_HASH，会建立一个列名到列的哈希表，这样就可以通过列名快速找到该列在fields中的位置。
- mem_root：该TABLE_SHARE对象内部使用的内存池。
- keynames：该表的索引名称以及索引类型。
- fieldnames：该表的列名以及列类型。
- tdc：该TABLE_SHARE对象的使用计数以及TABLE对象缓存。
- tabledef_version：表定义文件的版本信息。
- option_list：在执行create table命令时可以指定额外的选项，例如存储引擎类型、字符集、注释、行格式等信息。option_list为字符串格式的选项内容。
- option_struct：选项在MariaDB内部的表示。
- field：列信息。
- key_info：索引信息。
- blob_field：指示了哪些列的类型为blob。
- default_values：存储了各个列的默认值。
- comment：表的注释。在使用create table tb(...) comment="XXX"命令创建表时指定的表的注释。
- table_charset：字符串类型的列的字符集。
- table_cache_key：该成员的组成格式为data_base\0table_name\0，table cache以哈希表的形式缓存了数据库中各个表的TABLE对象，key就是table_cache_key，value为一个TABLE_SHARE对象和对应的多个TABLE对象。当我们需要获取TABLE对象时，可以将数据库和表名按照table_cache_key的格式生成key值，到table cache中快速获取对应的TABLE对象。
- db：该表所在数据库的名称。
- table_name：该表的名称。
- path：.frm文件的路径，可能包含~符号。
- normalized_path：转换后的.frm文件的路径，不包含~符号。
- min_rows和max_rows：在创建表时指定在该表中存储最少多少行数据和最多多少行数据，这不是一个硬性的限制，更像一个指示语句。
- avg_row_length：创建表时指定的表中行的平均长度的近似值，主要对包含长度可变类型的表进行该项设置。
- db_plugin：指向存储引擎插件。
- row_type：行存储格式，包括fixed、dynamic、compressed、compact，等等。
- transactional：该表是否是事务类型的表。
- page_checksum：是否支持页校验。
- key_block_size：创建表时指定的索引块的大小。

- stats_sample_pages：在进行统计估算的过程中取样的页数。
- fields：列的个数，包含虚拟列。
- stored_fields：列的个数，不包含虚拟列。
- rec_buff_length：行记录缓冲区的长度。
- null_fields：可以为NULL的列的个数。
- blob_fields：类型为blob的列的个数。
- varchar_fields：类型为varchar的列的个数。
- rowid_field_offset：列rowid的偏移量。在InnoDB存储引擎内部，每一行记录都含有一个隐藏的列rowid（行号，6字节）。通常情况下，InnoDB的表会在主键上生成聚簇索引，如果该表没有指定主键，会在unique索引上生成聚簇索引，如果unique索引也没有，就会在rowid列上生成聚簇索引。
- primary_key：主键的位置。
- error和open_errno：调用open_table_def函数读取.frm文件时发生的错误。
- vfields：虚拟列的个数。
- default_fields：包含默认值的列的个数。
- crypted：如果为true，表明.frm文件经过了加密。
- is_view：如果为true，说明当前表的类型为视图。
- deleting：如果为true，表明当前的表正在被删除。
- table_map_id：表在MariaDB内部的id，主要在基于行格式的复制时使用。
- tablespace：该表使用的表空间名称。

4

4.6 TABLE

类TABLE定义了数据库表的描述符。当我们执行查询语句的时候，需要打开语句中指定的表，这时就会创建一个TABLE实例。

下面我们给出类TABLE的定义：

```
// sql/table.h

struct TABLE
{
    TABLE_SHARE *s;                  // 指向对应的TABLE_SHARE对象
    handler *file;                   // 指向存储引擎对象
    TABLE *next, *prev;              // 用于实现TABLE对象的链表

private:
    // 使用同一个TABLE_SHARE对象的其他TABLE对象
    TABLE *share_next, **share_prev;
    friend struct TABLE_share;
```

```
public:
    THD *in_use;                     // 使用该TABLE对象的当前线程描述符
    Field **field;                   // 表对应的列
    uchar *record[2];                // 记录临时缓冲区

    key_map covering_keys;           // 能够覆盖当前查询的索引
    key_map keys_in_use_for_query;   // 当前查询可以使用的索引
    key_map keys_in_use_for_group_by; // 可以被group by使用的索引，这样就不需要进行额外的排序操作
    key_map keys_in_use_for_order_by; // 可以被order by使用的索引，这样就不需要进行额外的排序操作
    KEY  *key_info;                  // 该表的索引
    uint max_keys;                   // key_info数组的长度

    Field *next_number_field;        // 自增列
    Field **vfield;                  // 虚拟列

    Table_triggers_list *triggers;   // 该表上的触发器，如果没有则置为0

    String alias;                    // 表名或表别名
    query_id_t query_id;             // 打开并且使用该TABLE对象的查询

    ha_rows used_stat_records;       // 该表记录数的估计值，被查询优化器所使用
    ha_rows quick_condition_rows;    // 满足条件的记录数的预估值

    uint db_stat;

    uint derived_select_number;      // 如果是派生表，记录表的数

    bool force_index;
    bool force_index_order;
    bool force_index_group;

    bool key_read;                   // 如果为true，说明查询优化器已经知道只需要访问索引就可以获取
                                     // 所查询的数据（索引覆盖）
    bool no_keyread;
    bool created;                    // 如果当前表为临时表，该变量的值为true

    MEM_ROOT mem_root;
    GRANT_INFO grant;                // 该表的权限控制
    Filesort_info sort;              // 排序相关的信息，主要在order by和group by命令中使用

// 分区表信息
#ifdef WITH_PARTITION_STORAGE_ENGINE
    partition_info *part_info;
    bool all_partitions_pruned_away;
#endif

    MDL_ticket *mdl_ticket;          // 当前查询在该表上获取的mdl锁信息
};
```

下面我们给出TABLE类中各个成员的含义。

- s：指向对应的TABLE_SHARE对象。我们在4.5节已经介绍了，一个数据库表对应一个TABLE_SHARE对象。TABLE中保存的信息大多是和当前查询语句相关的，而TABLE_SHARE保存的大部分信息是与对应的数据库表相关的。
- file：指向表对应的存储引擎对象，通过该对象对数据库表进行操作。
- next和prev：用于实现TABLE链表。
- share_next和share_prev：指向使用同一个TABLE_SHARE对象的其他TABLE对象。
- in_use：当前使用该TABLE对象的线程。
- field：该表的列。
- record[2]：记录临时缓冲区。
- covering_keys：能够覆盖当前查询的索引。当查询的字段都在索引中的时候，就不需要访问对应的行数据，我们称之为索引覆盖。
- keys_in_use_for_query：当前查询可以使用的索引。
- keys_in_use_for_group_by：可以被group by命令使用的索引。因为B+树索引本身是按照索引列进行排序的，所以就不需要进行额外的排序操作。
- keys_in_use_for_order_by：可以被order by命令使用的索引，同上。
- key_info：该表的索引信息。
- max_keys：key_info数组的长度。
- next_number_field：指向自动增长的列，只可能有一个列是自动增长的。
- vfield：虚拟列。
- triggers：该表上的触发器。如果该表没有触发器，则将其置成0。
- alias：表名或表的别名。
- query_id：使用该TABLE对象查询的id。
- used_stat_records：该表记录数的一个估计值，被查询优化器所使用。
- quick_condition_rows：满足查询条件的记录数的估计值，主要在join算法中使用。
- db_stat：能够在该表上执行的操作。
- derived_select_number：如果是派生表，表示表的记录数。
- force_index：如果当前查询中使用了FORCE INDEX命令指定所用的索引，则该值为true。
- force_index_order：如果当前查询中使用了FORCE INDEX ORDER BY命令指定ORDER BY使用的索引，则该值为true。
- force_index_group：如果当前查询中使用了FORCE INDEX GROUP BY命令指定GROUP BY使用的索引，则该值为true。
- key_read和no_keyread：如果key_read的值为true，说明查询优化器知道当前查询满足索引覆盖条件，也就是只需要访问索引就能获取所需的数据。no_keyread的含义相反。
- created：如果该值为true，说明当前表为临时表。

- mem_root：该对象使用的内存池。
- grant：该表的权限信息。
- sort：主要用于group by和order by命令。
- mdl_ticket：当前表的mdl锁信息。

4.7　小结

本章中，我们介绍了MariaDB中一些基础的数据结构，这对想要了解MariaDB底层机制和想自行阅读MariaDB源代码的读者非常有帮助。

第5章

MariaDB线程池

5

我们知道，MySQL的传统连接模式是每连接每线程，MySQL会给每个连接上来的客户端分配一个单独的线程，该线程负责处理该客户端发来的所有命令。随着MySQL的连接数越来越多，MySQL的线程数也会相应地上升。MySQL默认的最大连接数是1024，也就是说当连接的客户端超过1024个之后，新来的客户端将连接不上MySQL的服务端。当然，可以通过调整参数max_connections来提高MySQL的最大连接数，但这又带来了其他问题：首先，每个线程会占用一定的系统资源，线程数越多消耗的系统资源也越多；其次，线程的创建和销毁是有一定开销的；最后，也是非常重要的一点，当线程数过多时，如果其中大部分线程都处于活跃状态，将会导致频繁的上下文切换，从而造成巨大的系统开销。

MariaDB从5.1开始引入了线程池技术来解决最大连接数限制问题以及过多线程带来的系统开销问题。线程池技术的本质就是线程共用，多个连接之间共享线程。

本章的内容主要包括：

- 线程池相关的参数
- 何时使用线程池
- 线程池的实现

5.1 线程池相关的参数

随着MariaDB版本的变更，线程池相关的实现也有所改动，线程池相关的参数也相应地发生了变化。本节中，我们将介绍MariaDB几个版本中的线程池。

5.1.1 MariaDB 5.1 和MariaDB 5.3 中的线程池

MariaDB 5.1和MariaDB 5.3中的线程池是静态的，线程池中的线程数从MariaDB启动时就是固定的。通过在配置文件中设置thread_pool_size参数来指定线程池中线程的个数：

```
thread_handling = pool_of_threads       # 线程模式
thread_pool_size = 8                    # 线程池大小
```

要想使用线程池，除了需要通过设置thread_pool_size参数来决定线程池中线程的个数外，还需要将thread_handling参数设置成pool_of_threads（表示使用线程池模型），该参数的默认值是one_thread_per_connection，就是传统的每连接每线程模式，它的另一个可选值为no_threads，表示使用一个线程来处理所有连接，这种处理连接的模型通常用于调试。

5.1.2　MariaDB 5.5 和MariaDB 10.0 中的线程池

在MariaDB 5.5及以后的版本中，线程池是动态的，线程池中的线程个数根据当前连接数的变化而在一定范围内浮动。MariaDB的线程池针对不同的操作系统使用了不同的实现，在Windows操作系统上MariaDB使用了系统原生的线程池，在类UNIX系统上，MariaDB实现了自己的线程池。你可以通过下列参数来设置线程池：

```
thread_handling = pool_of_threads       # 线程模式
thread_pool_size = 4                    # 线程池的分组数
thread_pool_min_threads = 4             # 仅在Windows上有效
thread_pool_max_threads = 16            # 线程池的最大线程数
thread_pool_stall_limit = 500           # timer线程的检查时间间隔，单位为毫秒
thread_pool_idle_timeout = 60           # worker线程的空闲超时时间，单位为秒
thread_pool_oversubscribe = 3           # 单个CPU核心上的“超频”线程数
```

下面我们来对这些参数进行详细的说明。

- thread_handling：和老版本的含义一致，设置为pool_of_threads表示使用线程池模型来处理连接。
- thread_pool_size：它表示的不再是线程池中的线程数，而是线程池的分组个数，默认值为当前机器CPU的核心数。在MariaDB 5.5和MariaDB 10.0中，线程池由多个分组组成，不同的连接被分配到不同的线程池分组中处理。实现分组的主要目的是为了把每个分组对应到每个CPU的核心上，保证每个分组中都有一个活跃的线程在运行，从而达到充分利用CPU资源的目的。
- thread_pool_min_threads：该参数只有在Windows系统上才有效，表示线程池中最少必须有多少个线程。
- thread_pool_max_threads：该参数用于控制线程池的最大线程数，防止线程池无限增长。
- thread_pool_stall_limit：在MariaDB的线程池中有一个单独的timer线程，用于定期检查线程池中是否有“停滞”的分组以及定期清理超时的客户端连接等工作。该参数表示的是多久可以算作停滞。我们将在5.3节中详细分析该参数是如何影响线程池的行为的。
- thread_pool_idle_timeout：当一个worker线程持续空闲了一段时间后会自动退出，以保证在业务空闲时减少线程池中的线程数。该参数的单位为秒。用户可以根据自己的业务场景来调整该参数的值，如果设置得太短，将会导致线程频繁地退出与创建；如果设置得太长，将会导致线程池中的线程数长时间不会下降。
- thread_pool_oversubscribe：该参数用于控制单个CPU核心上“超频”的线程数，如果单个CPU核心上活跃的线程数为1，我们认为它没有“超频”。MariaDB将线程池设计为由多

个分组组成，每个线程池分组对应于一个CPU的核（默认配置），如果能够保证每个线程池分组活跃的线程数控制在一定范围内，就能很好地利用系统资源。该参数的默认值是3，尽量不要修改该参数。

5.2　何时使用线程池

在有大量短查询的业务场景下，在每连接每线程的模式中，一方面过多的连接数很容易到达连接数最大值的限度，另一方面过多的活跃线程导致频繁的上下文切换带来不可忽视的系统开销。在这种情况下，线程池能够很好地发挥作用。由于都是短查询，不会有某个连接长时间占用线程池中的线程，所以几乎不会影响客户端请求的响应时间，并且随着连接数的增加，线程池中的线程数都被控制在一定范围内，减轻了系统的压力。

但在有大量长查询的业务场景下不适合使用线程池，在这种场景下，由于长查询占据了线程池的所有线程，导致线程池出现效率低下的情况，客户端甚至不能进行连接。当然，你可以通过配置extra_port参数来配置每连接每线程模式的额外端口，用户可以使用该端口进行连接。

5.3　线程池的实现

本节中，我们从源代码的角度来分析MariaDB的线程池是如何工作的，这里我们参考的是MariaDB 10.0的实现，它与MariaDB 5.5中线程池的实现区别不是很大。

图5-1很好地展示了线程池的整体结构。线程池由多个分组组成，每个分组有一个任务队列，用于存储待处理的任务。listener线程监听当前分组中的所有连接，将需要处理的连接添加到任务队列中，worker线程循环处理任务队列中的任务。此外，线程池中还包含一个timer线程，用于检查分组是否处于“停滞”状态以及定期清理过期的客户端连接。

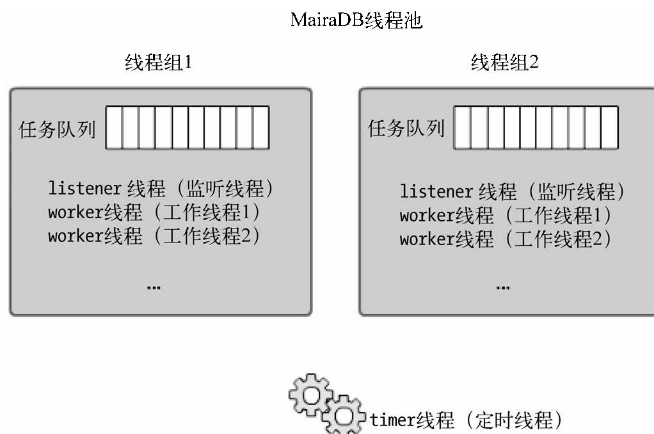


图5-1　线程池模型

5.3.1 线程池相关的数据结构

首先，我们来看一下线程池相关的几个数据结构的定义。

1. connection_t

结构体connection_t代表客户端的连接，其中包含了客户端连接的所有信息：

```
// sql/threadpool_unix.cc

struct connection_t                          // 代表一个客户端的连接
{
    THD *thd;                                // 连接的所有上下文，包括网络套接字等
    thread_group_t *thread_group;            // 所属的线程池分组
    connection_t *next_in_queue;             // next_in_queue和prev_in_queue用于实现队列
    connection_t **prev_in_queue;
    ulonglong abs_wait_timeout;              // 客户端的超时时间
    bool logged_in;                          // 该连接是否进行了登录验证
    bool bound_to_poll_descriptor;           // 该连接是否加入到了监听poll
    bool waiting;
};
```

下面给出了connection_t中各个成员的具体含义。

- thd：包含了连接的所有上下文信息，包括该连接的网络套接字等。
- thread_group：MariaDB的线程池被设计成包含多个分组，该成员指向该连接所在的线程池分组。
- next_in_queue：指向下一个connection_t结构。
- prev_in_queue：和next_in_queue一起实现connection_t的队列，在结构connection_queue_t中使用到。
- abs_wait_timeout：客户端的超时时间。在后面介绍MariaDB线程池timer线程的时候会讲到，timer线程会定期清理掉超时的客户端。
- logged_in：如果该值为true，表示该连接已经进行了登录验证；如果为false，说明该连接还没有进行登录验证，在执行该连接发来的命令前需要进行登录验证。
- bound_to_poll_descriptor：每个线程池分组内部的所有连接都会被加入到分组对应的poll内进行网络事件的监听。如果该参数为true，说明该连接的网络套接字已经加入到了poll内进行监听。在Linux操作系统上，使用的是epoll来进行网络事件的监听。
- waiting：如果该参数为true，表示该连接正在等待锁、条件变量、IO的完成，等等。

2. thread_group_t

结构thread_group_t代表了线程池中的一个分组，其定义如下：

```
// sql/threadpool_unix.cc

struct thread_group_t                         // 代表线程池的一个分组
{
    mysql_mutex_t mutex;
    connection_queue_t queue;                 // 任务队列
    worker_list_t waiting_threads;            // 睡眠的worker线程列表
    worker_thread_t *listener;                // listener线程
    pthread_attr_t *pthread_attr;             // 创建线程的属性
    int  pollfd;                              // poll的描述符
    int  thread_count;                        // 当前分组中的线程数
    int  active_thread_count;                 // 当前分组中的活跃线程数
    int  connection_count;                    // 当前分组中包含的连接数
    int io_event_count;                       /* 该线程池分组在这个时间段的网络事件数，timer线程会
                                                 使用该信息判断一个分组是否处于“停滞”状态 */
    int queue_event_count;                    /* 该线程池分组在这个时间段消费的任务数，timer线程会
                                                 使用该信息判断一个分组是否处于“停滞”状态 */
    ulonglong last_thread_creation_time;      // 最近一次创建线程的时间
    int  shutdown_pipe[2];                    // 用于关闭线程池分组
    bool shutdown;                            // true代表关闭
    bool stalled;                             // true代表当前线程池分组处于“停滞”状态
};
```

5

下面列出了thread_group_t中各个成员的含义。

- mutex：由于thread_group_t的成员会被多个线程操作，所以必须有锁保护。
- queue：线程池分组内的待处理任务会被加入到该队列中，等待worker线程处理。
- waiting_threads：当worker线程处理完任务后，如果任务队列没有更多的任务，worker线程会进入睡眠状态，进入睡眠状态的worker线程会被加入到waiting_threads中，等待有新的任务到来时被唤醒，或者由于超时而自动退出。
- listener：在正常情况下，每个线程池分组内部都有一个listener线程对组内的所有连接的网络事件进行监听。listener成员指向listener线程。
- pthread_attr：我们介绍的线程池版本是针对MariaDB 10.0的，该版本的线程池是动态的，所以在运行时会有线程退出和创建。pthread_attr保存了创建线程时所需要的线程属性。
- pollfd：线程池分组中所有连接的网络套接字都会被加入到poll中，用来监听连接的网络事件，pollfd就是该poll的描述符。例如，在Linux平台上，使用的是epoll，那么pollfd就是epoll_create函数返回的描述符。
- thread_count：该线程池分组中的总线程数，包括listener线程和所有worker线程，不管该worker线程是处于活跃状态还是睡眠状态。
- active_thread_count：活跃的worker线程数，也就是正在消费任务队列中任务的worker线程。
- connection_count：当前线程池分组的连接数。

- io_event_count：这个时间段监听到的网络事件数。timer线程会使用该变量来判断线程池分组是否处于“停滞”状态，这在5.3.7节中会进一步介绍。
- queue_event_count：这个时间段worker线程消费的任务数。timer线程会使用该变量来判断线程池分组是否处于“停滞”状态，这在5.3.7节中会进一步介绍。
- last_thread_creation_time：最近一次创建线程的时间。
- shutdown_pipe：用于关闭线程池分组。
- shutdown：如果为true，表示该线程池分组已经关闭。
- stalled：如果为true，表示该线程池分组处于“停滞”状态，这在5.3.7节中会进一步介绍。

3. worker_thread_t

结构worker_thread_t代表一个worker线程或者一个listener线程，其定义如下：

```
// sql/threadpool_unix.cc

struct worker_thread_t
{
    ulonglong  event_count;           // 该worker线程处理的请求数
    thread_group_t *thread_group;     // 该worker线程所在的线程池分组
    worker_thread_t *next_in_list;    // next_in_list和prev_in_list实现链表
    worker_thread_t **prev_in_list;

    mysql_cond_t  cond;
    bool          woken;              // true表示线程处于唤醒状态
};
```

下面给出了worker_thread_t各个成员的含义。

- event_count：如果worker_thread_t代表的是一个worker线程，event_count则表示该worker线程处理的总请求数。
- thread_group：该线程所在的线程池分组。
- next_in_list和prev_in_list：用于实现链表，在worker_list_t中使用。
- cond：事件通知的条件变量。
- woken：如果为true，表示线程处于唤醒状态。

5.3.2　线程池的初始化

接下来，我们从mysqld启动时线程池的初始化作为切入点，按照程序的执行流来分析MariaDB线程池的工作机制。mysqld的入口函数是sql/mysqld.cc中的main函数，该函数进行一系列的初始化工作，其中包括调用tp_init函数对线程池进行初始化，然后调用handle_connections_sockets函数循环处理新来的连接。

tp_init函数的主要功能是对MariaDB的线程池进行初始化，其代码如下：

```
// sql/threadpool_unix.cc

1. bool tp_init()
2. {
       // 为thread_group_t分配内存
3.     all_groups = (thread_group_t *)my_malloc(sizeof(thread_group_t) * threadpool_size,
4.         MYF(MY_WME|MY_ZEROFILL));
5.     if (!all_groups)
6.     {
7.         threadpool_size= 0;
8.         DBUG_RETURN(1);
9.     }

       // 初始化
10.    threadpool_started= true;
11.    scheduler_init();
12.    for (uint i= 0; i < threadpool_size; i++)
13.    {
14.        // 对thread_group_t的成员进行默认的设置
15.        thread_group_init(&all_groups[i], get_connection_attrib());
16.    }

       //为各个线程池分组分配poll的描述符
17.    tp_set_threadpool_size(threadpool_size);
18.    if(group_count == 0)
19.    {
20.        sql_print_error("Can't set threadpool size to %d",threadpool_size);
21.        DBUG_RETURN(1);
22.    }

       // 开启timer线程，此时线程池中仅有的一个线程
23.    pool_timer.tick_interval= threadpool_stall_limit;
24.    start_timer(&pool_timer);
25.    DBUG_RETURN(0)
26.}
```

第3行到第9行代码为线程池各个分组分配内存，all_groups是一个全局变量，管理线程池的所有分组。

第10行到第22行代码用于对线程池分组进行初始化，并且为每个线程池分组创建一个poll描述符，用于监听线程池分组内所有连接的网络事件。

第23行到第25行代码用于开启timer线程。至此，线程池的初始化工作已经完成。目前，线程池中只有一个线程，那就是timer线程。

5.3.3　添加连接到线程池

上面我们已经介绍了mysqld.cc的main函数在执行完一系列的初始化后，会进入handle_connections_sockets函数，该函数循环监听服务端口（3306）的事件，通过调用tp_add_connection函数将新来的连接加入到线程池中（如果我们配置的是线程池模式）。

下面给出了tp_add_connection函数的定义：

```
// sql/threadpool_unix.cc

1. void tp_add_connection(THD *thd)
2. {
3.     threads.append(thd);
4.     mysql_mutex_unlock(&LOCK_thread_count); // 解除之前获取的锁

5.     connection_t *connection= alloc_connection(thd);
6.     if (connection)
7.     {
           // 将当前连接分配到对应的线程池分组中
8.         thd->event_scheduler.data= connection;
9.         thread_group_t *group= &all_groups[thd->thread_id%group_count];
10.        connection->thread_group=group;

11.        mysql_mutex_lock(&group->mutex);
12.        group->connection_count++;
13.        mysql_mutex_unlock(&group->mutex);

           // 将连接加入到任务队列中，如果当前分组没有活跃的线程，唤醒空闲的线程进行工作或者创建新的
           // 线程处理任务
14.        queue_put(group, connection);
15.    }
16.    else
17.    {
18.       threadpool_remove_connection(thd);
19.    }
20.    DBUG_VOID_RETURN;
21.}

22.static void queue_put(thread_group_t *thread_group, connection_t *connection)
23.{
24.    mysql_mutex_lock(&thread_group->mutex);

       // 将新来的连接加入到任务队列中
25.    thread_group->queue.push_back(connection);

       // 如果当前线程池分组中没有活跃的线程，则唤醒空闲的线程或者创建新的线程处理任务
26.    if (thread_group->active_thread_count == 0)
27.        wake_or_create_thread(thread_group);
```

```
28.    mysql_mutex_unlock(&thread_group->mutex);

29.    DBUG_VOID_RETURN
30.}
```

第5行代码为新来的连接创建了一个connection_t对象。

第8行到第13行代码将该连接加入到对应的线程池分组中，并且增加该线程池分组总连接数的计数。

第14行代码调用queue_put将该连接加入到任务队列中，让线程池分组中的worker线程去处理。前面我们已经提到，当线程池初始化好之后，内部只有一个timer线程，没有listener线程或者worker线程。如果这是我们接收的第一个连接，那么当代码执行到queue_put函数的第27行时，会为该线程池分组新建一个worker线程处理这个新来的连接。

5.3.4 worker线程

worker线程的主要任务是处理任务队列中待处理的任务。下面我们给出worker线程的入口函数worker_main的定义，本节中我们将按照代码的执行流程来分析worker线程是如何工作的：


```
// sql/threadpool_unix.cc

1. static void *worker_main(void *param)
2. {
       // 初始化
3.     worker_thread_t this_thread;
4.     pthread_detach_this_thread();
5.     my_thread_init();

6.     thread_group_t *thread_group = (thread_group_t *)param;

7.     mysql_cond_init(key_worker_cond, &this_thread.cond, NULL);
8.     this_thread.thread_group= thread_group;
9.     this_thread.event_count=0;

       // 循环处理任务
10.    for(;;)
11.    {
12.        connection_t *connection;
13.        struct timespec ts;
14.        set_timespec(ts,threadpool_idle_timeout);
15.        connection = get_event(&this_thread, thread_group, &ts);
16.        if (!connection)
17.            break;
18.        this_thread.event_count++;
19.        handle_event(connection);
```

```
20.    }

       // 线程退出
21.    mysql_cond_destroy(&this_thread.cond);

22.    bool last_thread;
23.    mysql_mutex_lock(&thread_group->mutex);
24.    add_thread_count(thread_group, -1);
25.    last_thread= ((thread_group->thread_count == 0) && thread_group->shutdown);
26.    mysql_mutex_unlock(&thread_group->mutex);

       // 如果是分组的最后一个线程，并且线程池分组的shutdown标志是开启的，则销毁线程池分组
27.    if (last_thread)
28.        thread_group_destroy(thread_group);

29.    my_thread_end();
30.    return NULL;
31.}
```

第3行到第9行代码是一些初始化工作。

第4行代码将当前线程的状态设置成detached，这样当线程退出时，会自动释放所有的资源。

第10行到第20行代码是处理任务的主要流程。首先调用get_event函数获取待处理的任务，然后调用handle_event函数处理任务，处理完之后继续获取任务，依次循环执行。handle_event函数对于没有登录的连接会进行登录验证，然后才会将该连接的网络套接字加入到poll中，以便监听该连接的后续命令；对于已经登录的连接，读取网络数据，执行客户端发送过来的命令。

第16行代码中，当get_event获取的待处理任务返回空时，跳出任务处理循环，执行函数的后半部分，退出worker线程。

第24行代码，将当前线程池分组的总线程数减1。

第25行代码，判断当前线程是否是当前分组的最后一个线程。

第27到第28行代码中，如果是分组中的最后一个线程，并且线程池分组的shutdown标志为true，则执行销毁线程池分组的操作。

5.3.5　get_event函数

get_event函数负责从任务队列获取一个任务，返回给worker线程处理。接下来我们看一下get_event函数是如何获取任务的，在什么情况下会返回空，从而导致worker线程退出：

```
// sql/threadpool_unix.cc

1. connection_t *get_event(worker_thread_t *current_thread,
```

```
2.      thread_group_t *thread_group,  struct timespec *abstime)
3. {
4.      connection_t *connection = NULL;
5.      int err=0;

        // 需要锁保护，因为要对thread_group的成员进行操作
6.      mysql_mutex_lock(&thread_group->mutex);
7.      DBUG_ASSERT(thread_group->active_thread_count >= 0);

8.      for(;;)
9.      {
            /*
              如果当前线程池分组的活跃线程过多，会导致“超频”
              参数thread_pool_oversubscribe在此发挥作用
            */
10.         bool oversubscribed = too_many_threads(thread_group);
11.         if (thread_group->shutdown)
12.             break;

13.         if (!oversubscribed)
14.         {
                // 如果任务队列中有任务，直接获取并返回
15.             connection = queue_get(thread_group);
16.             if(connection)
17.                 break;
18.         }

            /*
              如果当前线程池分组没有listener线程，让当前worker线程成为listener线程，
              监听当前线程池分组的网络事件
            */
19.         if(!thread_group->listener)
20.         {
21.             thread_group->listener= current_thread;
22.             thread_group->active_thread_count--;
23.             mysql_mutex_unlock(&thread_group->mutex);

                // 监听当前线程池分组的事件
24.             connection = listener(current_thread, thread_group);

25.             mysql_mutex_lock(&thread_group->mutex);
26.             thread_group->active_thread_count++;
27.             thread_group->listener= NULL;
28.                 break;
29.             }

             /*
               worker线程在进入睡眠状态之前会尝试看一下poll中有没有事件需要处理，
               timeout参数设置成0表示如果poll中没有事件立刻返回，不等待
```

5

```
          */
30.       if (!oversubscribed)
31.       {
32.           native_event nev;
33.           if (io_poll_wait(thread_group->pollfd,&nev,1, 0) == 1)
34.           {
35.               thread_group->io_event_count++;
36.               connection = (connection_t *)native_event_get_userdata(&nev);
37.               break;
38.           }
39.       }

          // 进入睡眠状态，把当前线程加入睡眠线程列表中
40.       current_thread->woken = false;
41.       thread_group->waiting_threads.push_front(current_thread);

42.       thread_group->active_thread_count--;
          // abstime传进来的是参数thread_pool_idle_timeout的值
43.       if (abstime)
44.       {
45.           err = mysql_cond_timedwait(&current_thread->cond, &thread_group->mutex, abstime);
45.       }
47.       else
48.       {
49.           err = mysql_cond_wait(&current_thread->cond, &thread_group->mutex);
50.       }

          /*
            线程被唤醒，要么是被listener线程调用wake_thread函数唤醒的，
            要么是因为超时而唤醒的
          */
51.       thread_group->active_thread_count++;

          // 非wake_thread唤醒的，需要自己从睡眠列表中删除
52.       if (!current_thread->woken)
53.       {
54.           thread_group->waiting_threads.remove(current_thread);
55.       }

          /*
            如果线程是因为超时而唤醒的，跳出for循环，此时connection的内容为NULL，
            将会导致该worker线程退出。
            如果是被listener线程调用wake_thread唤醒的，说明有任务需要处理，获取任务
          */
56.       if (err)
57.           break;
58.   }

59.   thread_group->stalled= false;
```

```
60.     mysql_mutex_unlock(&thread_group->mutex);

61.     DBUG_RETURN(connection);
62.}
```

第6行代码获取锁保护，这是因为后续我们会对线程池分组的成员进行操作。

第10行代码用于判断当前线程池分组的活跃线程数是否太多，是否达到了参数thread_pool_oversubscribe设定的“超频”标准。

第11行代码用于判断线程池分组是否处于关闭状态，如果是，则跳出循环，此时变量connection的值为空，将导致worker线程退出。

第13行到第18行代码说明，在当前线程池分组没有超频的情况下，如果任务队列中有待处理的任务，直接获取并且返回，返回的任务会传给handle_event函数处理，此处可参考worker_main函数的第19行。

如果任务队列中暂时没有任务，程序将继续往下执行。第19行到第29行代码说明，如果当前的线程池分组中没有listener线程，将当前worker线程转变成listener线程，监听当前线程池分组中所有连接的事件。如果当前线程池分组中已经有listener线程，程序将继续执行，此时任务队列中没有待处理的任务，并且线程池分组也有专门的listener线程在监听可能发生的事件，worker线程接下来即将进入睡眠状态。

第30行到第39行代码说明，worker线程在进入睡眠状态之前会尝试看看poll中是否有需要处理的事件。之后的第40行到第50行代码说明，该worker线程进入睡眠状态，等待有新任务到来时被唤醒或者由于睡眠太久超时而退出。

当程序执行到第51行时，说明worker线程被唤醒。可能是由于listener线程监听到新的任务，需要worker线程进行处理，或者是由于mysqld主线程接收到新的连接，需要worker线程进行处理，或者是worker线程睡眠的时间超过了thread_pool_idle_timeout设定的值主动醒来的。对于前面两种情况，worker线程从任务队列中获取待处理的任务进行处理；对于后一种情况，说明目前业务比较空闲，worker线程主动退出，使线程池的线程数下降。

5.3.6　listener线程

通常情况下，每个线程池分组都有一个单独的listener线程用于监听当前分组的所有网络事件。

我们在前面看到，当worker线程发现任务队列中没有待处理的任务，并且当前的线程池分组中没有listener线程时，worker线程会调用listener函数，摇身一变，成为了listener线程。下面我们通过分析listener函数的实现来了解listener线程的主要工作：

// sql/threadpool_unix.cc

```
1. static connection_t * listener(worker_thread_t *current_thread, thread_group_t *thread_group)
2. {
3.     DBUG_ENTER("listener");
4.     connection_t *retval= NULL;

5.     for(;;)
6.     {
7.         native_event ev[MAX_EVENTS];
8.         int cnt;

9.         if (thread_group->shutdown)
10.            break;

           // 监听事件
11.        cnt = io_poll_wait(thread_group->pollfd, ev, MAX_EVENTS, -1);

12.        if (cnt <=0)
13.        {
14.            DBUG_ASSERT(thread_group->shutdown);
15.            break;
16.        }

17.        mysql_mutex_lock(&thread_group->mutex);

18.        if (thread_group->shutdown)
19.        {
20.            mysql_mutex_unlock(&thread_group->mutex);
21.            break;
22.        }

           /*
             统计这一时间段新增的事件数，
             timer线程将使用该数据判断线程池是否处于“停滞”状态
           */
23.        thread_group->io_event_count += cnt;

            /*
              当监听到网络事件后，listener线程根据任务队列的情况来决定是参与处理事件，
              还是将所有的任务交给worker线程去处理
            */
24.         bool listener_picks_event= thread_group->queue.is_empty();
25.         for(int i=(listener_picks_event)?1:0; i < cnt ; i++)
26.         {
                // 将待处理的任务加入到任务队列中
27.             connection_t *c= (connection_t *)native_event_get_userdata(&ev[i]);
28.             thread_group->queue.push_back(c);
29.         }
```

```
30.         if (listener_picks_event)
31.         {
                // listener线程处理第一个事件，listener线程转变成worker线程
32.             retval= (connection_t *)native_event_get_userdata(&ev[0]);
33.             mysql_mutex_unlock(&thread_group->mutex);
34.             break;
35.         }

            // 如果当前线程池分组没有活跃的worker线程，则唤醒或者新建一个worker线程进行工作
36.         if(thread_group->active_thread_count==0)
37.         {
38.             if(wake_thread(thread_group))
39.             {
40.                 if(thread_group->thread_count == 1)
41.                 {
42.                     create_worker(thread_group);
43.                 }
44.             }
45.         }

46.         mysql_mutex_unlock(&thread_group->mutex);
47.     }

48.     DBUG_RETURN(retval);
49.}
```

listener线程的核心任务在代码的第11行，用于监听线程池分组中所有连接的事件。在Linux系统中，函数io_poll_wait就是对epoll_wait函数的封装。

第23行代码说明，当监听到事件时，将监听到的事件数统计到线程池分组的io_event_count中去，该变量统计的是这一时间段该线程池分组监听到的网络事件数，timer线程将会使用该统计数据来评估该线程池分组是否处于“停滞”状态。

普通的理解是，当listener线程监听到事件时，只需将监听到的事件加入到任务队列中等待worker线程进行处理，然后继续监听。MariaDB的线程池稍微做了一些改进，listener线程在监听到事件后，会检查当前任务队列里是否有任务，如果没有任务，listener线程变成worker线程来处理这些任务，当任务队列中有未处理的任务时，则唤醒或者创建worker线程来处理这些事件。这部分功能对应于代码的第24行到第47行。

listener线程为什么会转变成worker线程呢？当监听到事件时，listener线程从poll中醒来，状态从waiting变成running，如果listener线程仅仅是将事件加入到待处理任务队列中，然后唤醒或者创建worker线程来处理这些事件，接着继续进入监听状态，这种机制一方面是没有很好地利用listener线程的时间片（刚刚从waiting状态变为running状态，又要变回waiting状态），另一方面是唤醒或者创建一个worker线程来处理任务总是没有直接使用当前线程处理来得快。为什

么当任务队列不为空时，listener线程就不会转换成worker线程来处理事件呢？这是因为如果任务队列不为空，说明目前的网络事件比较频繁，下一次网络事件可能在很短的时间内就会到来，为了提高事件的响应速度，listener线程会继续执行监听任务。

至此，我们发现MariaDB线程池是动态的，这个动态不仅仅表现为线程池的线程数会在一定范围内变化，而且线程池中的listener线程和worker线程不是一成不变的，在某些情况下会相互转换角色。

5.3.7 timer线程

MariaDB的线程池中存在一个timer线程，该线程的主要工作是检查线程池分组是否处于“停滞”状态以及定期清理掉超时的客户端连接。

timer线程的入口函数是timer_thread，其代码如下所示：

```
// sql/threadpool_unix.cc

1. static void *timer_thread(void *param)
2. {
3.     uint i;
4.     pool_timer_t *timer=(pool_timer_t *)param;

5.     my_thread_init();
6.     DBUG_ENTER("timer_thread");
7.     timer->next_timeout_check= ULONGLONG_MAX;
8.     timer->current_microtime= microsecond_interval_timer();

9.     for(;;)
10.    {
11.        struct timespec ts;
12.        int err;

           // 定时器
13.        set_timespec_nsec(ts,timer->tick_interval*1000000);
14.        mysql_mutex_lock(&timer->mutex);
15.        err= mysql_cond_timedwait(&timer->cond, &timer->mutex, &ts);
16.        if (timer->shutdown)
17.        {
18.            mysql_mutex_unlock(&timer->mutex);
19.            break;
20.        }

           //定期执行以下任务
21.        if (err == ETIMEDOUT)
22.        {
23.            timer->current_microtime= microsecond_interval_timer();
```

```
            //检查线程池分组是否处于“停滞”状态
24.         for (i= 0; i < threadpool_size; i++)
25.         {
26.             if(all_groups[i].connection_count)
27.                  check_stall(&all_groups[i]);
28.         }

            //清除超时的客户端连接
29.         if (timer->next_timeout_check <= timer->current_microtime)
30.             timeout_check(timer);
31.     }
32.     mysql_mutex_unlock(&timer->mutex);
33.  }

34.  mysql_mutex_destroy(&timer->mutex);
35.  my_thread_end();
36.  return NULL;
37.}

38.void check_stall(thread_group_t *thread_group) // 检查线程池分组是否处于“停滞”状态
39.{
40.  if (mysql_mutex_trylock(&thread_group->mutex) != 0)
41.  {
        // 正在执行一些操作，不要打扰
42.     return;
43.  }

     /*
       没有listener线程，并且从本次检查到上次检查这段时间，该线程池分组没有网络事件，
       可能由于listener线程正在执行某个长查询，需要唤醒或启动一个worker线程（转化成listener）
     */
44.  if (!thread_group->listener && !thread_group->io_event_count)
45.  {
46.     wake_or_create_thread(thread_group);
47.     mysql_mutex_unlock(&thread_group->mutex);
48.     return;
49.  }

     // 重置统计变量io_event_count
50.  thread_group->io_event_count= 0;

     // 如果队列不为空，而且这段时间没有从队列消费任务， 则唤醒或启动一个worker线程
51.  if (!thread_group->queue.is_empty() && !thread_group->queue_event_count)
52.  {
53.     thread_group->stalled= true;
54.     wake_or_create_thread(thread_group);
55.  }
```

```
    // 重置统计变量queue_event_count
56.    thread_group->queue_event_count= 0;

57.    mysql_mutex_unlock(&thread_group->mutex);
58.}
```

timer线程的主要工作在第21行到第31行代码，具体如下。

- 定期检查线程池分组是否处于“停滞”状态。
- 定期清理超时的客户端连接。

第13到第20行代码用于启动定时器。这里MariaDB使用了条件变量的超时机制来实现定时器功能。

第24行到第28行代码，调用check_stall函数，对线程池的所有分组进行检查，检查其是否处于“停滞”状态。

第29行到第30行代码用于清理掉超时的客户端连接。

如何判断一个线程池分组是否处于“停滞”状态呢？我们可以进入check_stall函数一探究竟。

第44行到第49行代码说明如果当前线程池分组中没有listener线程，并且从上一次检查到本次检查之间没有监听到任何网络事件，统计变量io_event_count为0，很有可能之前的listener线程转变成了worker线程正在执行一个长查询还没有返回，这个时候需要唤醒或者创建一个worker线程，该线程处理完队列中的剩余任务后会自动变成listener线程。

第50行代码用于重置统计变量io_event_count。

第51行到第55行代码说明，如果任务队列有待处理的任务，并且从上一次检查到本次检查之间没有从任务队列消费任何的任务，很有可能现有的worker线程正在执行长查询，导致任务队列发生了拥塞，这个时候需要唤醒或者创建新的worker线程来分担处理的压力。

第56行代码用于重置统计变量queue_event_count。

5.4 小结

本章中，我们详细介绍MariaDB线程池的工作原理以及具体实现。线程池由多个分组组成，每个分组由一个任务队列、一个listener线程以及多个worker线程组成。线程池中还存在一个timer线程，它用于检查线程池分组的状态以及定期清理掉过期的客户端连接。

线程池技术在解决MariaDB最大连接数问题以及大量线程带来的系统开销方面发挥了重要的作用。

第 6 章 二进制日志binlog

MariaDB/MySQL提供了4种不同的日志，分别是错误日志（error log）、普通日志（general log）、慢日志（slow log）以及二进制日志（binlog）。错误日志记录了系统启动、运行以及停止过程中遇到的一些问题；普通日志记录了MariaDB/MySQL执行的所有语句以及语句开始执行的时间等信息，用户可以选择性地打开它；慢日志记录了MariaDB/MySQL所有慢查询的相关信息；而二进制日志则以事件的形式记录了MariaDB/MySQL的库表结构以及表数据的所有变更信息。本章中，我们将详细讲解二进制日志所涉及的一些内容。

本章的内容主要包括：

- 简介
- binlog的使用
- binlog事件
- 清理binlog
- binlog_cache_mngr结构
- mysqlbinlog工具
- 使用binlog进行恢复

6.1 简介

binlog是记录所有数据库表结构变更（例如CREATE、ALTER TABLE...）以及表数据修改（INSERT、UPDATE、DELETE...）的二进制日志。

从宏观上来看，binlog由一系列binlog文件和一个index文件组成。数据库的所有变更信息以事件的形式记录在binlog文件中，index文件记录了当前使用了哪些binlog文件。这里有必要重申一下binlog和binlog文件的区别，binlog表示整个binlog体系，包括所有的binlog文件和index文件，而binlog文件表示一个记录了数据库修改信息的具体文件。在本书的剩余部分，我们将延续这里的定义。

binlog文件以一个4字节的常量作为开头（标识这是一个binlog文件），后面跟着一系列的binlog事件。对于不同的binlog格式，相同语句记录在binlog文件中的事件也有所不同。

binlog不会记录那些没有任何修改的语句，例如select和show等查询命令，你可以通过查询普通日志来查看MySQL执行过的所有语句。

6.1.1　binlog的作用

binlog有两个非常重要的功能：复制和备份恢复。

- **复制**：在MariaDB/MySQL的主从结构中，主库的binlog记录了主库的所有更改操作，从库通过读取主库的binlog，在本地重放获取的binlog，这样从库就拥有和主库相同的数据，达到复制的目的。
- **备份恢复**：binlog记录了数据库的所有更改信息，所以当MariaDB/MySQL发生崩溃的时候，能够以最近备份点作为起点，然后执行在备份点之后产生的binlog中的所有事件，实现数据库最大可能的恢复。

除了上面介绍的两个作用外，binlog对于事务存储引擎的崩溃恢复也有非常重要的作用。在开启binlog的情况下，MariaDB/MySQL将采用事务的两阶段提交协议。当MariaDB/MySQL server或者系统发生崩溃时，事务在存储引擎内部的状态可能为prepared和commit两种。对于prepared状态的事务，是进行提交操作还是进行回滚操作，这时需要参考binlog：如果事务在binlog中存在，那么将其提交；如果在binlog中不存在，那么将其回滚，这样就保证了数据在主库和从库之间的一致性。

6.1.2　index文件

为了管理所有的binlog文件，MariaDB/MySQL额外创建了一个*base-name*.index文件，它按顺序记录了MariaDB/MySQL使用的所有binlog文件。如果你想自定义index文件的名称，可以设置`--log-bin-index=file`参数。千万不要在mysqld运行的时候手动修改index文件的内容，这样会使mysqld产生混乱。图6-1演示binlog的构成。

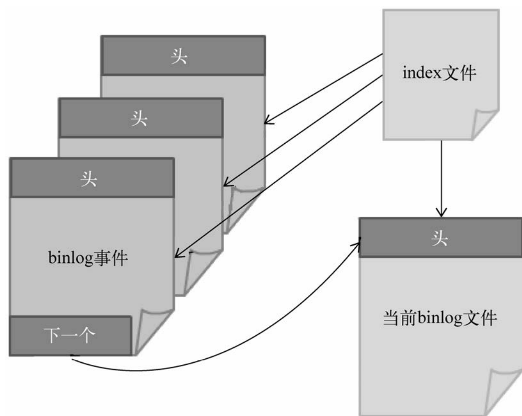


图6-1　binlog的组成

6.2 binlog 的使用

本节中我们将从使用的角度来介绍binlog的相关内容，主要包括如何开启binlog、如何选择binlog的格式以及binlog几个比较重要的参数。

6.2.1 开启binlog

如果想要开启binlog，可以在MariaDB/MySQL的配置文件中添加`log-bin[=base-name]`的配置，也可以在启动mysqld的时候携带`--log-bin[=base-name]`参数。如果`base-name`是一个绝对路径，那么MariaDB/MySQL会将你的binlog文件存储在`base-name`指定的文件夹下。如果`base-name`不是一个绝对路径，那么MariaDB/MySQL会以`base-name`作为前缀来命名你的binlog文件，并将其存储在配置的数据目录下。如果没有指定`base-name`，那么MySQL会以hostname-bin作为binlog文件的默认前缀，其中hostname是你的主机名。如果你指定的`base-name`含有后缀名，MariaDB/MySQL会忽略该后缀名。binlog文件以一个自增的数字作为后缀，例如my-binlog.000026：

```
[mysqld]
log-bin=base-name              #开启binlog，并且指定binlog文件名的前缀
```

6.2.2 选择binlog的格式

通过配置`BINLOG_FORMAT`参数的值，可以选择binlog的格式，也可以通过执行`SET GLOBAL BINLOG_FORMAT="{STATEMENT|ROW|MIXED}"`命令在MariaDB/MySQL运行的时候动态更改binlog的格式。参数`BINLOG_FORMAT`有3个可选的值：`STATEMENT`、`ROW`和`MIXED`，分别代表3种不同的binlog格式：

```
[mysqld]
BINLOG_FORMAT=ROW                #指定binlog的格式为ROW
```

1. binlog的3种格式

binlog的3种格式如下所示。

- `STATEMENT`：顾名思义，`STATEMENT`格式的binlog记录的是数据库上执行的原生SQL语句。
- `ROW`：这种格式的binlog记录的是数据表的行是怎样被修改的。
- `MIXED`：如果设置了这种格式，MariaDB/MySQL会在一些特定的情况下自动从`STATEMENT`格式切换到`ROW`格式。例如，包含`uuid`等不确定性函数的语句，引用了系统变量的语句，等等。

关于binlog格式，有个容易混淆的地方，那就是binlog文件总是由一系列binlog事件组成的，对数据库进行同样的修改时，如果设置的binlog格式不同，则仅仅是记录的事件类型不同，并不是binlog的“格式”发生了什么变化。通常情况下，在MariaDB/MySQL运行的时候改变binlog的

格式是可行的（通过调用SET GLOBAL BINLOG_FORMAT="{STATEMENT|ROW|MIXED}"语句，然后退出当前会话，重新建立一个连接才会生效），但不推荐，因为某些情况下会发生错误。例如，语句执行过程中包含临时表，因为临时表只会记录在STATEMENT格式下，而在ROW格式下是不会记录的。

2. 各种binlog格式的优缺点

STATEMENT格式的显著优点就是产生的binlog文件比ROW格式的binlog文件小，这在使用binlog文件备份数据库的时候会占用较小的磁盘空间。由于STATEMENT格式的binlog记录的是原生SQL语句，所以可以通过mysqlbinlog工具很容易读懂其中的内容。但对于不确定性的事件，使用STATEMENT格式是有问题的。例如，当语句中使用了USER、UUID、SYSDATE等不确定性函数时，在主从复制结构中，从节点读取主节点的binlog事件然后进行重放，可能会导致从数据库和主数据库的内容不一致。

ROW格式的binlog产生的binlog文件通常相对较大，但解决了STATEMENT格式的binlog在含有不确定事件的时候导致主从数据不一致的问题，因为ROW格式的binlog是按照表的行数据修改情况来记录的。ROW格式被认为是最安全的数据库复制方式。需要注意的是，在使用了ROW格式的binlog之后，执行的DDL语句和FLUSH系列语句还是会以文本的形式记录下来（事件类型为query_event）。当使用的binlog格式为ROW时，如果一条UPDATE语句匹配的行数很多，这个时候会向binlog写入大量的数据。

6.2.3　binlog的相关参数

1. max_binlog_size参数

可以通过配置max_binlog_size参数来限定单个binlog文件的大小。如果当前binlog文件的大小达到了参数指定的阈值，会创建一个新的binlog文件作为当前活跃的binlog文件，后续所有对数据库的修改都会记录在新的binlog文件中。

对于binlog文件的大小，有个需要注意的地方是，binlog文件可能会大于max_binlog_size参数设定的阈值。由于一个事务所产生的所有事件必须记录在同一个binlog文件中，所以即使binlog文件的大小达到max_binlog_size参数指定的大小，也要等到当前事务的所有事件全部写入到binlog文件中才能切换，这样就会出现binlog文件的大小大于max_binlog_size参数指定大小的情况。

2. binlog过滤器

拥有root权限的用户可以通过执行SET sql_log_bin=0命令来禁用当前会话（连接）的binlog功能，该会话执行的所有操作都不会记录在binlog中。

MariaDB/MySQL也可以通过配置binlog_do_db和binlog_ignore_db参数选择需要记录binlog的数据库：

```
[mysqld]
binlog_do_db=db1
binlog_ignore_db=db2
```

3. sync_binlog参数

默认情况下（sync_binlog=0），binlog文件在每次写入内容后是不会立刻持久化到磁盘上的，具体的持久化操作交给了操作系统来做。这样，当操作系统崩溃的时候，对binlog文件进行的修改可能会丢失，从而造成binlog文件的数据丢失和不一致。为了避免发生这种情况，可以将sync_binlog配置成1，这样在将事务写入到binlog文件中之后立即执行fsync操作将binlog文件的修改同步到磁盘上。但这样会降低MariaDB/MySQL的性能，因为fsync是个昂贵的系统调用。你也可以将sync_binlog配置成一个整数*N*，指定在写入*N*个事务之后才执行一次fsync操作，这时如果系统崩溃了，binlog最多只会丢掉*N*个事务的数据。在第7章中，我们将介绍MariaDB采用binlog group commit技术提高了在大量并发事务的情况下事务的提交速度。所以，在通常情况下，我们应该总是将sync_binlog参数设置为1。

6.3 binlog 事件

MariaDB/MySQL binlog以事件的形式来记录数据库的变更情况。通过执行show binlog events in "*binlog-file*"命令来查看指定binlog文件中的事件，如图6-2所示。

```
mysql> show binlog events in "mysql-bin.000026";
+------------------+------+-------------+-----------+-------------+------------------------------------------
| Log_name         | Pos  | Event_type  | Server_id | End_log_pos | Info
+------------------+------+-------------+-----------+-------------+------------------------------------------
| mysql-bin.000026 |    4 | Format_desc |         1 |         120 | Server ver: 5.6.14-debug-log, Binlog ver: 4
| mysql-bin.000026 |  120 | Query       |         1 |         242 | use `python`; create table abcd(a int(11), b
| mysql-bin.000026 |  242 | Query       |         1 |         316 | BEGIN
| mysql-bin.000026 |  316 | Table_map   |         1 |         368 | table_id: 75 (python.abcd)
| mysql-bin.000026 |  368 | Write_rows  |         1 |         412 | table_id: 75 flags: STMT_END_F
| mysql-bin.000026 |  412 | Xid         |         1 |         443 | COMMIT /* xid=25 */
| mysql-bin.000026 |  443 | Query       |         1 |         517 | BEGIN
| mysql-bin.000026 |  517 | Table_map   |         1 |         569 | table_id: 75 (python.abcd)
| mysql-bin.000026 |  569 | Write_rows  |         1 |         620 | table_id: 75 flags: STMT_END_F
| mysql-bin.000026 |  620 | Xid         |         1 |         651 | COMMIT /* xid=39 */
```

图6-2 show binlog events命令

执行show binlog events命令后，我们可以获取事件类型、事件在文件中的位置等信息。如果binlog的格式为STATEMENT，还能看出具体执行的SQL语句。

6.3.1 binlog事件格式

binlog事件由公有事件头（common-header）、私有事件头（post-header）和事件体（body）3部分组成，如图6-3所示。所有的事件都包含公有事件头。在固定版本的binlog中，公有事件头的长度和格式是固定的。根据事件类型的不同，某些binlog事件还包含私有事件头。binlog事件的最后一部分就是事件体，根据事件类型的不同，事件体的格式和包含的信息也各不相同，也有一些

事件没有事件体，例如stop_event仅仅包含一个公有事件头。

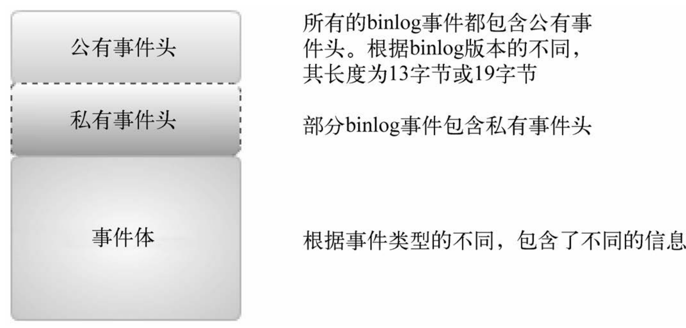


图6-3　binlog事件的组成

所有的binlog事件都以一个13或者19字节的公有事件头开始，其中包含了该事件发生的时间、事件类型、事件长度以及server-id等信息。公有事件头的定义如下：

```
长度          字段
4             timestamp
1             event type
4             server-id
4             event size
if binlog-version > 1:
4             log pos
2             flags
```

图6-4给出了公有事件头的组成情况，图中的虚线部分表示只有版本大于1的binlog才包含。

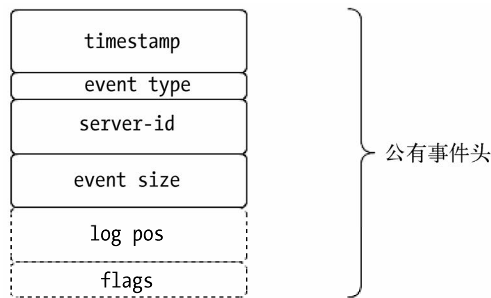


图6-4　binlog公有事件头

timestamp字段包含了该事件的开始执行时间，event type字段指明了该事件的类型，server_id字段是产生该事件的MariaDB/MySQL服务器的server-id，event size字段标识了该事件的长度，包括公有事件头、私有事件头和事件体3部分的长度。

对于版本大于1的binlog，公有事件头的长度是19字节，其中log pos字段指示了下一个事件

在binlog文件中的位置。flags字段则包含了一些额外的信息。例如，如果FORMAT_DESCRIPTION_EVENT事件的flags中包含了LOG_EVENT_BINLOG_IN_USE_F标志，表明当前binlog正在使用，MariaDB在启动的时候能够通过检查binlog的第一个事件FORMAT_DESCRIPTION_EVENT的该标志来判断binlog是否能正常关闭，如果非正常关闭表明MariaDB需要进行异常恢复操作。

6.3.2　binlog事件类型

图6-5简单展示了binlog文件中用到的一些事件，下面我们将对图中列出的事件一一进行介绍。当然，还有一些binlog事件这里没有列出，有兴趣的读者可以自行查阅相关资料。

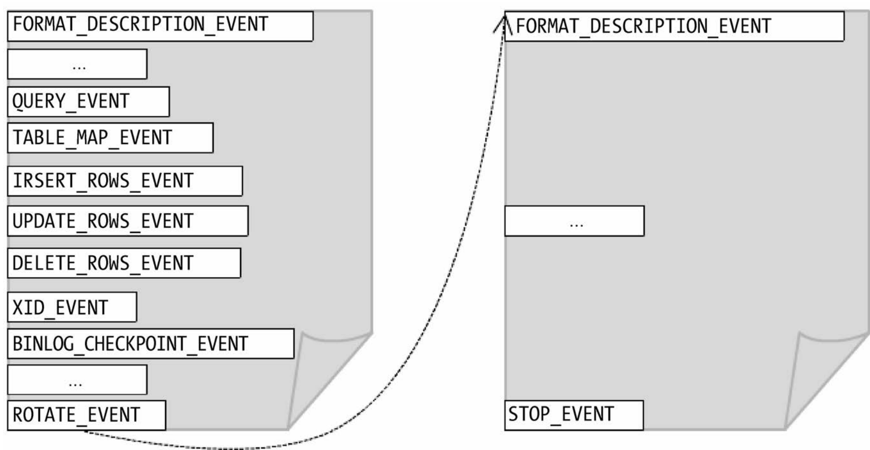


图6-5　binlog由一系列的事件组成

1. FORMAT_DESCRIPTION_EVENT

格式描述事件是binlog version 4中为了取代之前版本中的START_EVENT_3事件而引入的。它是所有binlog文件中的第一个事件，该事件在一个binlog文件中仅会出现一次。MariaDB/MySQL根据FORMAT_DESCRIPTION_EVENT事件的定义来解析binlog中的其他事件。FORMAT_DESCRIPTION_EVENT由公有事件头和事件体两部分组成。事件体的定义如下：

```
FORMAT_DESCRIPTION_EVENT body
2                          binlog-version
string[50]                 mysql-server version
4                          create timestamp
1                          event header length
string[p]                  event type header lengths
```

表6-1给出了FORMAT_DESCRIPTION_EVENT事件体各个字段的具体含义。

表6-1　FORMAT_DESCRIPTION_EVENT事件体各个字段的含义

字　段	长　度	位　置	说　明
binlog-version	2字节	事件体	binlog版本
mysql-server version	50字节	事件体	服务器版本
create timestamp	4字节	事件体	该字段指明该binlog文件的创建时间。如果该binlog是由于切换而产生的（执行flush logs命令或者binlog文件的大小达到了max_binlog_size参数指定的值），那么将该字段设置为0
event header length	1字节	事件体	19
event type header lengths	数组	事件体	该字段是一个数组，记录了所有事件的私有事件头的长度

2. QUERY_EVENT

QUERY_EVENT以文本的形式来记录信息。当binlog的格式是statement的时候，执行的SQL语句都记录在QUERY_EVENT中，如图6-6所示。

```
mysql> show binlog events in "mysql-bin.000029";
+------------------+-----+-------------+-----------+-------------+-------------------------------------------------
| Log_name         | Pos | Event_type  | Server_id | End_log_pos | Info
+------------------+-----+-------------+-----------+-------------+-------------------------------------------------
| mysql-bin.000029 |   4 | Format_desc |         1 |         120 | Server ver: 5.6.14-debug-log, Binlog ver: 4
| mysql-bin.000029 | 120 | Query       |         1 |         203 | BEGIN
| mysql-bin.000029 | 203 | Query       |         1 |         329 | use `python`; insert into abcd values(13, "this is about
| mysql-bin.000029 | 329 | Xid         |         1 |         360 | COMMIT /* xid=147 */
| mysql-bin.000029 | 360 | Query       |         1 |         443 | BEGIN
| mysql-bin.000029 | 443 | Query       |         1 |         564 | use `python`; update abcd set b="new values" where a < 10
| mysql-bin.000029 | 564 | Xid         |         1 |         595 | COMMIT /* xid=149 */
```

图6-6　QUERY_EVENT

QUERY_EVENT由公有事件头、私有事件头和事件体3部分组成，相关定义如下：

```
QUERY_EVENT post-header：
4                  slave_proxy_id
4                  execution time
1                  schema length
2                  error-code
if binlog-version ≥ 4:
2                  status-vars length

QUERY_EVENT body：
string[$len]       status-vars
string[$len]       schema
1                  [00]
string[EOF]        query
```

表6-2　QUERY_EVENT各个字段的含义

字　段	长　度	位　置	说　明
slave_proxy_id	4字节	私有事件头	某些查询可能会创建临时表，而这些临时表仅仅在当前的连接或会话中有效。为了区分不同连接或会话中的临时表，slave_proxy_id存储了不同连接或会话的线程id

（续）

字　段	长　度	位　置	说　明
execution time	4字节	私有事件头	查询从开始执行到记录到binlog所花的时间，单位为秒
schema length	1字节	私有事件头	schema字符串长度
error-code	2字节	私有事件头	错误码
status-vars length	2字节	私有事件头	status-vars长度
status-vars	status-vars length	事件体	status-vars字段是以键/值对的形式保存起来的一系列由SET命令设置的上下文信息，例如是否开启autocommit
schema	schema length	事件体	当前选择的数据库
query	取决于查询的长度	事件体	query的文本格式，里面存储的可能是BEGIN、COMMIT字符串或者原生的SQL语句，等等

QUERY_EVENT类型的事件通常在以下几种情况中使用。

- 事务开始时，在binlog中记录一个类型为QUERY_EVENT的BEGIN事件。
- 在STATEMENT格式的binlog中，具体执行的SQL语句保存在QUERY_EVENT事件中。
- 对于ROW格式的binlog，所有的DDL操作以文本的格式记录在QUERY_EVENT事件中，如图6-7所示。

```
| mysql-bin.000034 | 321 | Query      |  1 |  395 | BEGIN
| mysql-bin.000034 | 395 | Table_map  |  1 |  447 | table_id: 72 (python.test)
| mysql-bin.000034 | 447 | Write_rows |  1 |  491 | table_id: 72 flags: STMT_END_F
| mysql-bin.000034 | 491 | Xid        |  1 |  522 | COMMIT /* xid=23 */
| mysql-bin.000034 | 522 | Query      |  1 |  645 | use `python`; alter table test add column
```

图6-7　QUERY_EVENT事件记录ROW格式中的DDL语句

3. ROWS_EVENT

对于STATEMENT格式的binlog，所有的增删改操作的原生SQL语句都记录在QUERY_EVENT中，而ROW格式的binlog以ROWS_EVENT的形式记录对数据库数据的修改。ROWS_EVENT分为3种，WRITE_ROWS_EVENT、UPDATE_ROWS_EVENT和DELETE_ROWS_EVENT，分别对应于INSERT、UPDATE和DELETE语句。WRITE_ROWS_EVENT包含了要插入的数据；UPDATE_ROWS_EVENT不仅包含了行修改后的值，也包含了行修改前的值；DELETE_ROWS_EVENT仅仅包含了需要删除行的主键值/行号。

3种ROWS_EVENT的格式很类似，下面给出它们的定义：

```
row event post-header:
if post_header_len == 6 {
    4                   table-id
} else {
    6                   table-id
}
2                       flags
if version == 2 {
```

```
    2                    extra-data-len
    string.var_len       extra-data
}

row event body:
lenenc_int               number of columns
string.var_len           columns-present-bitmap1, length: (num of columns+7)/8
if UPDATE_ROWS_EVENTv1 or v2 {
    string.var_len           columns-present-bitmap2, length: (num of columns+7)/8
}

rows:
string.var_len           nul-bitmap, length (bits set in 'columns-present-bitmap1'+7)/8
string.var_len           value of each field as defined in table-map
if UPDATE_ROWS_EVENTv1 or v2 {
    string.var_len           nul-bitmap, length (bits set in 'columns-present-bitmap2'+7)/8
    string.var_len           value of each field as defined in table-map
}
... repeat rows until event-end
```

图6-8给出了rows-event各个字段的排列情况。

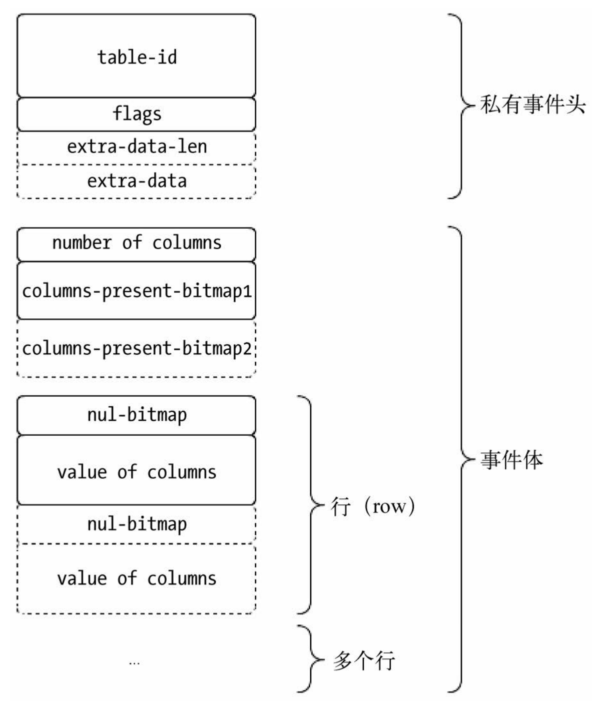


图6-8 ROWS_EVENT的组成

表6-3列出了ROWS_EVENT各个版本的情况。

表6-3 ROWS_EVENT版本情况

版 本	包含的事件	特 性
v0	update_rows_eventv0 write_rows_eventv0 delete_rows_eventv0	/
v1	update_rows_eventv1 write_rows_eventv1 delete_rows_eventv1	为UPDATE_ROWS_EVENT事件添加了行修改前的值
v2	update_rows_eventv2 write_rows_eventv2 delete_rows_eventv2	在v1的基础上添加了extra_data

表6-4列出了ROWS_EVENT各个字段的具体含义。

表6-4 ROWS_EVENT各个字段的含义

字 段	长 度	位 置	说 明
table-id	4或6字节	私有事件头	该ROWS_EVENT对应的表id
flags	2字节	私有事件头	可以包含以下信息：该ROWS_EVENT是否是语句的最后一个事件，是否需要进行外键约束的检查，针对InnoDB的二级索引是否需要进行唯一性检查，该ROWS_EVENT是否包含了完整一行的数据，也就是说覆盖了所有列
extra-data-len	2字节	私有事件头	extra-data的长度
extra-data	extra-data-len	私有事件头	仅在版本2的ROWS_EVENT中存在。用于携带额外的数据，主要目的是用于扩展
number of columns	1、3、4或9字节	事件体	表的列数
columns-present-bitmap1	(number of columns+7)/8	事件体	以位图的形式指示了该ROWS_EVENT包含了哪些列的数据
columns-present-bitmap2	(number of columns+7)/8	事件体	对于新版本（v1和v2）的UPDATE_ROWS_EVENT事件，不仅包含列修改后的值，还包含列修改前的值
nul-bitmap	(bits set in(columns-present- bitmap1)+7)/8	事件体	columns-present-bitmap中为空的列，MariaDB会以NULL或者列对应的默认值补全
value of columns	取决于列的值	事件体	列的数据

4. TABLE_MAP_EVENT

在ROW格式的binlog文件中，每个ROWS_EVENT事件之前都有一个TABLE_MAP_EVENT，用于描述表的内部id和结构定义，如图6-9所示。

```
mysql> show binlog events in "mysql-bin.000030";
+------------------+-----+-------------+-----------+-------------+------------------------------
| Log_name         | Pos | Event_type  | Server_id | End_log_pos | Info
+------------------+-----+-------------+-----------+-------------+------------------------------
| mysql-bin.000030 |   4 | Format_desc |         1 |         120 | Server ver: 5.6.14-debug-log, B
| mysql-bin.000030 | 120 | Query       |         1 |         194 | BEGIN
| mysql-bin.000030 | 194 | Table_map   |         1 |         246 | table_id: 75 (python.abcd)
| mysql-bin.000030 | 246 | Write_rows  |         1 |         307 | table_id: 75 flags: STMT_END_F
| mysql-bin.000030 | 307 | Xid         |         1 |         338 | COMMIT /* xid=170 */
| mysql-bin.000030 | 338 | Query       |         1 |         412 | BEGIN
| mysql-bin.000030 | 412 | Table_map   |         1 |         464 | table_id: 75 (python.abcd)
| mysql-bin.000030 | 464 | Update_rows |         1 |         660 | table_id: 75 flags: STMT_END_F
| mysql-bin.000030 | 660 | Xid         |         1 |         691 | COMMIT /* xid=194 */
| mysql-bin.000030 | 691 | Query       |         1 |         765 | BEGIN
| mysql-bin.000030 | 765 | Table_map   |         1 |         817 | table_id: 75 (python.abcd)
| mysql-bin.000030 | 817 | Delete_rows |         1 |         871 | table_id: 75 flags: STMT_END_F
| mysql-bin.000030 | 871 | Xid         |         1 |         902 | COMMIT /* xid=198 */
```

图6-9　TABLE_MAP_EVENT

TABLE_MAP_EVENT的定义如下：

```
table_map_event post-header:
if post_header_len == 6 {
    4          table id
} else {
    6          table id
}
2              flags

table_map_event body:
1              schema name length
string         schema name
1              [00]
1              table name length
string         table name
1              [00]
lenenc-int     column-count
string.var_len [length=$column-count] column-type-def
lenenc-str     column-meta-def
n              NULL-bitmap, length: (column-count + 7) / 8
```

TABLE_MAP_EVENT各个字段的含义如表6-5所示。

表6-5　TABLE_MAP_EVENT各个字段的含义

字段名称	字段长度	所属部分	说　明
table-id	4或6字节	私有事件头	表id
flags	2字节	私有事件头	标志位，暂时未使用
schema name	schema name length	事件体	表所在数据库的名称
table name	table name length	事件体	表名
column-count	1、3、4或9字节	事件体	表的列数
column-type-def	column-count	事件体	列的类型
column-meta-def	长度取决于列的类型	事件体	列的元信息
null-bitmap	(column-count+7)/8	事件体	以位图的形式记录可以为NULL的列

5. XID_EVENT

当提交事务时，不管是STATEMENT还是ROW格式的binlog，都会添加一个XID_EVENT事件作为事务的结束。该事件记录了该事务的id。在MariaDB/MySQL进行崩溃恢复的时候，根据事务在binlog中的提交情况来决定是否提交存储引擎中状态为prepared的事务。

XID_EVENT的格式很简单，事件体仅仅包含一个xid字段：

```
xid_event body:
8               xid
```

6. BINLOG_CHECKPOINT_EVENT

该事件是MariaDB引入的新事件，主要用于崩溃恢复。在两阶段事务提交过程中，当MariaDB崩溃时，我们需要根据binlog中事务的提交情况来决定是否提交存储引擎内部状态为prepared的事务。为了减少恢复过程中需要读取的binlog文件数，当某个binlog文件内部的所有事务都在存储引擎内部提交了，这时我们会在binlog中写入一个BINLOG_CHECKPOINT_EVENT事件。执行崩溃恢复的过程中，我们会根据所读取的BINLOG_CHECKPOINT_EVENT来决定哪些binlog文件是可以不用扫描的。

下面给出了MariaDB中BINLOG_CHECKPOINT_EVENT事件的实现，该事件指明了崩溃恢复时扫描binlog的起始位置：

```
class Binlog_checkpoint_log_event: public Log_event
{
public:
    char *binlog_file_name;
    uint binlog_file_len;
    ...
};
```

7. ROTATE_EVENT

当binlog文件的大小达到max_binlog_size参数设置的值时或者执行flush logs命令时，binlog会发生切换，这时会在当前使用的binlog文件末尾添加一个ROTATE_EVENT事件，将下一个binlog文件的名称记录在该事件中，如图6-10所示。

```
mysql> flush logs;
Query OK, 0 rows affected (0.27 sec)

mysql> show binlog events in "mysql-bin.000032";
+------------------+-----+-------------+-----------+-------------+------------------------------
| Log_name         | Pos | Event_type  | Server_id | End_log_pos | Info
+------------------+-----+-------------+-----------+-------------+------------------------------
| mysql-bin.000032 |   4 | Format_desc |         1 |         120 | Server ver: 5.6.14-debug-log,
| mysql-bin.000032 | 120 | Query       |         1 |         194 | BEGIN
| mysql-bin.000032 | 194 | Table_map   |         1 |         246 | table_id: 75 (python.abcd)
| mysql-bin.000032 | 246 | Write_rows  |         1 |         301 | table_id: 75 flags: STMT_END_F
| mysql-bin.000032 | 301 | Xid         |         1 |         332 | COMMIT /* xid=239 */
| mysql-bin.000032 | 332 | Rotate      |         1 |         379 | mysql-bin.000033;pos=4
+------------------+-----+-------------+-----------+-------------+------------------------------
```

图6-10　ROTATE_EVENT

ROTATE_EVENT的定义如下：

```
rotate_event post-header:
if binlog-version > 1 {
    8      position
}

rotate_event body:
string[p]      name of the next binlog
```

8. STOP_EVENT

当MariaDB/MySQL停止时，会在当前binlog文件的结尾写入一个STOP_EVENT事件来表示数据库停止，如图6-11所示。

```
| mysql-bin.000033 | 374 | Xid        |       1 |       405 | COMMIT /* xid=2
| mysql-bin.000033 | 405 | Stop       |       1 |       428 |
+------------------+-----+------------+---------+-----------+---------------
```

图6-11　STOP_EVENT

STOP_EVENT仅仅包含一个公有事件头，没有私有事件头和事件体部分，因为只需要在公有事件头的event type字段指定为STOP_EVENT就可以了，不需要携带额外的信息。

6.3.3　binlog事件的实现

在MariaDB中，所有的事件类都是从Log_event类继承而来的，该类定义了所有binlog事件的共同属性和函数。图6-12给出的是MariaDB中各个事件类的继承关系。

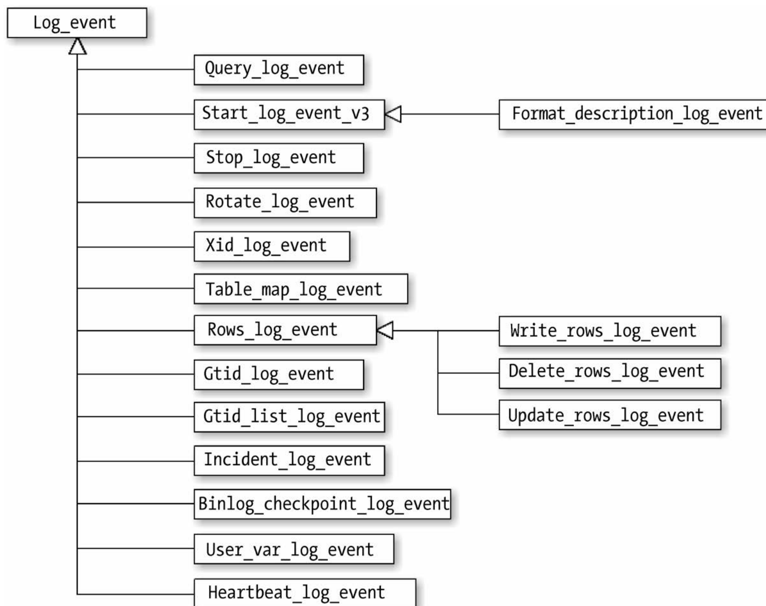


图6-12　binlog事件类的继承图

6.4 清理 binlog

随着MariaDB/MySQL的运行，产生的binlog越来越多，当binlog占用的磁盘空间比较多的时候，就需要清理过期的或者不再需要的binlog文件。

6.4.1 手动清理binlog

通常，有两种方式来手动清理binlog，一种是使用MariaDB/MySQL提供的purge命令，一种是使用系统自带的rm命令。purge命令的定义如下：

```
purge {binary | master} logs to "binlog-file-name"
purge {binary | master} logs before "datetime-expr"
```

其中第一种形式的purge命令的作用是将*binlog-file-name*之前的所有binlog文件清理掉，而第二种形式的purge命令的作用是将最后修改时间早于*datetime-expr*的binlog文件清理掉。

使用rm命令手动清理binlog的流程如下。

(1) 确保你的MariaDB/MySQL处于停止状态。
(2) 使用rm命令按顺序删除binlog文件。
(3) 修改index文件，把已经删除的binlog文件从index文件中删除。

在使用rm命令清理binlog时，首先应该确保MariaDB/MySQL处于停止状态，因为我们要手动修改index文件。其次需要注意的是，index文件是按顺序记录使用了哪些binlog文件，所以使用rm命令来删除binlog文件时，一定要按照其在index文件中的顺序来清理，否则会出现问题。

6.4.2 自动清理binlog

除了手动清理binlog外，还有一种自动清理binlog的方法。在启动MariaDB/MySQL server的时候，携带expire_logs_days=N参数（0≤N≤99），或者在配置文件中添加expire_logs_days=N选项，这样MariaDB/MySQL server只会保存N天的binlog，过期的binlog文件会被自动清理掉。

自动清理的具体实现是：当binlog文件发生切换或者MariaDB server启动时，MariaDB会遍历index文件，找到第一个“最后修改时间”在*N*天内的binlog文件，然后将该binlog文件之前的所有binlog文件删除掉。

6.4.3 purge命令的实现

这里我们先来看一下purge {binary | master} logs to "*binlog-file*"命令的具体执行过程。该命令会调用MYSQL_BIN_LOG::purge_logs函数来完成具体的工作，下面我们给出该函数的具体实现：

```
// sql/log.cc

1. int MYSQL_BIN_LOG::purge_logs(const char *to_log,
                              bool included,     //如果为true，to_log也会被清理掉
                              bool need_mutex,   //是否需要加锁
                              bool need_update_threads,
                              ulonglong *decrease_log_space)
2. {
3.     int error= 0;
4.     bool exit_loop= 0;
5.     LOG_INFO log_info;
6.     THD *thd= current_thd;

7.     if (need_mutex)
8.         mysql_mutex_lock(&LOCK_index);

       // 检查to_log是否存在
9.     if ((error=find_log_pos(&log_info, to_log, 0 /*no mutex*/)))
10.    {
11.        goto err;
12.    }

       // 创建一个purge_index_file，用于保存待删除的binlog
13.    if ((error= open_purge_index_file(TRUE)))
14.    {
15.        goto err;
16.    }

       // 遍历index文件，将可以删除的binlog添加到purge_index_file中
17.    if ((error=find_log_pos(&log_info, NullS, 0 /*no mutex*/)))
18.        goto err;
19.    while ((strcmp(to_log,log_info.log_file_name) || (exit_loop=included)) &&
           can_purge_log(log_info.log_file_name))
20.    {
21.        if ((error= register_purge_index_entry(log_info.log_file_name)))
22.        {
23.            goto err;
24.        }

25.        if (find_next_log(&log_info, 0) || exit_loop)
26.            break;
27.    }

       // 将purge_index_file写入到磁盘
28.    if ((error= sync_purge_index_file()))
29.    {
30.        goto err;
31.    }

       // 更新index文件，将待删除的binlog文件名从中删除掉
```

```
32.    if ((error=update_log_index(&log_info, need_update_threads)))
33.    {
34.        goto err;
35.    }

36.err:
       // 删除所有待删除的binlog文件
37.    if (is_inited_purge_index_file() &&
           (error= purge_index_entry(thd, decrease_log_space, FALSE)))
38.        sql_print_error("MSYQL_BIN_LOG::purge_logs failed to process registered files"
               " that would be purged.");

       // 删除purge_index_file文件
39.    close_purge_index_file();

40.    if (need_mutex)
41.        mysql_mutex_unlock(&LOCK_index);

42.    DBUG_RETURN(error);
43.}
```

先看purge_logs函数中各个参数的含义。

- to_log：将会删除to_log之前的所有binlog文件。
- included：如果该参数为true，那么to_log本身也会被删除掉。
- need_mutex：如果该参数为true，在整个执行过程中需要加锁。
- need_update_threads：由于index文件有修改，binlog文件在index中的偏移量发生了变化，需要通知各个线程进行更新。
- decrease_log_space：更新binlog占用的磁盘空间。

第9行代码检查to_log在index文件中是否存在，避免非法的purge操作。

第13行代码创建一个purge_index_file，用于保存待删除的binlog文件名称。

第17行到第27行代码遍历index文件，将to_log之前（或者包括to_log）能够被删除（没有使用）的binlog文件加入到purge_index_file中。在遍历的过程中，如果某个binlog文件正在使用（例如，该binlog文件正在被复制线程所使用，或者该binlog文件为当前活跃的binlog文件），那么将会跳出循环，purge操作仅仅会删除被使用的binlog文件之前的所有其他binlog文件。

第28 ~ 31行代码，将purge_index_file的内容写入到磁盘。

第32 ~ 35行代码更新index文件的内容，将待删除的binlog文件从index文件中删除掉。

至此，如果系统或者MariaDB server进程崩溃，也不会发生index文件和binlog文件不一致的情况。在MariaDB server重新启动的时候会检查purge_index_file是否存在，如果存在，会将其中

6

记录的binlog文件删除。

第37行代码将记录在purge_index_file中所有待删除的binlog文件删除掉。

第39行代码将purge_index_file文件删除掉。

purge {binary | master} logs before "datetime-expr"命令的执行过程也是类似的，该命令首先寻找第一个“最后修改时间”大于datetime-expr的binlog文件，然后以该binlog文件名作为to_log参数调用MYSQL_BIN_LOG::purge_logs函数。

6.5　binlog_cache_mngr 结构

对于非事务的存储引擎，所有的修改会立刻写入到binlog文件中。对于事务的存储引擎，事情会稍微复杂一点。因为一个事务可能包含多条语句，如果所有的修改立刻写入到binlog文件中，那么当用户需要回滚该事务的时候就会陷入麻烦之中。MariaDB使用了binlog_cache_mngr结构来缓存一条事务产生的所有修改。如果用户执行提交操作，就将binlog_cache_mngr的内容写入到binlog文件中；如果用户执行回滚操作，将会丢弃binlog_cache_mngr内的修改，这样就保证binlog文件的内容和数据库的修改保持一致，如图6-13所示。

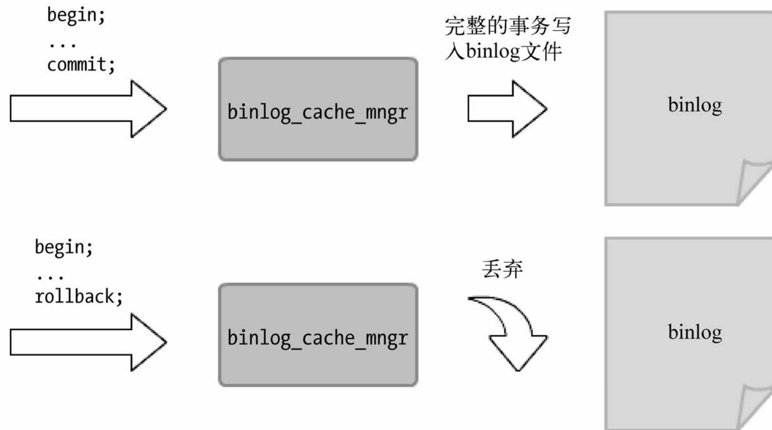


图6-13　binlog_cache_mngr

6.6　mysqlbinlog 工具

MariaDB/MySQL的binlog以二进制的形式来描述数据库是如何被修改的，mysqlbinlog工具可以将binlog中事件包含的信息以文本的形式打印出来。

mysqlbinlog的执行方式很简单：mysqlbinlog [options] binlog-file ...

下面我们截取了mysqlbinlog的一段输出：

```
# at 421
#140320 8:57:52 server id 1 end_log_pos 525 CRC32 0x0d2b0848 Query thread_id=12 exec_time=0 error_ code=0
SET TIMESTAMP=1395277072/*!*/;
DELETE FROM test_table WHERE a=2
/*!*/;
```

第一行的at指明了该事件在binlog文件中的位置。第二行描述了该事件开始的执行时间，执行了多长时间，执行该事件的线程号等信息。最后一行给出了该事件所执行的SQL语句。

mysqlbinlog的输出是“可执行”的。将mysqlbinlog的输出作为mysql命令的输入，就能重放binlog中记录的修改，这对于MariaDB/MySQL的崩溃恢复是很有价值的。

6.7　使用 binlog 进行恢复

当MariaDB/MySQL崩溃之后，可能会造成数据丢失和不一致。如果我们定期对数据库进行备份，就可以以最近备份点为基础，在此之上重放这段时间binlog中记录的修改，把我们的数据库恢复到崩溃前的状态。

通过binlog进行恢复主要有以下两步。

(1) 使你的数据库恢复到最近备份点的状态。

(2) 执行`mysqlbinlog your-bin-log | mysql -u root -p`，将binlog中记录的修改反映到数据库中。

如果你有多个binlog文件，第二步需要执行多次，并且要保证是在同一个会话中按顺序执行的，不然将会导致数据不一致。

6.8　小结

在阅读完这一章之后，你应该对MariaDB/MySQL的binlog有了一个全面的认识。这里我们再来回顾下其中一些重要的内容。

- MariaDB/MySQL的binlog由一系列binlog文件和一个index文件组成：binlog文件以事件的形式记录了MariaDB/MySQL所有的修改，index文件记录了MariaDB/MySQL所使用的所有binlog文件。不要在MariaDB/MySQL运行时手动修改index文件。
- binlog的格式可以设置为STATEMENT、ROW或者MIXED。STATEMENT格式的binlog以文本的格式记录了执行的修改语句，它比ROW格式产生的binlog文件小。ROW格式的binlog解决了STATEMENT格式的binlog在含有不确定事件时候导致的主从数据不一致的问题。

- `sync_binlog`参数的选择会影响到MariaDB/MySQL的性能和binlog数据的完整性，在实际应用中，我们应该尽量把`sync_binlog`的值设置成1。
- 通过 `purge {binary|master} logs`命令可以清理过期的binlog。
- binlog文件是由一系列事件组成的,这些事件描述了MariaDB/MySQL数据库的内容是如何被修改的。
- 我们可以通过mysqlbinlog工具来查看binlog中记录的事件，并且可以通过重新执行binlog中记录的修改来达到数据库崩溃恢复的目的。

第 7 章

binlog group commit技术

在关系数据库中，为了满足ACID中的D（持久化）属性，也就是说事务提交并且成功返回给客户端之后，必须保证该事务的所有修改都持久化了，无论是在数据库程序崩溃的情况下或者是数据库所在的服务器发生宕机或者断电的情况下，都必须保证数据不能丢失。这就要求数据库在事务提交过程中调用fsync或fdatasync将数据持久化到磁盘。fsync是一个昂贵的系统调用，对于普通的磁盘，每秒只能完成几百次的fsync操作。很明显，fsync将会限制每秒钟提交的事务数，成为关系数据库的瓶颈。

对于MariaDB/MySQL，这种情况变得更加糟糕。在开启binlog的情况下，为了保证主库和从库之间数据的一致性，MariaDB/MySQL使用了事务的两阶段提交协议。在这种情况下，为了满足数据的持久化需求，一个事务的提交最多会导致3次fsync操作。

为了提高MariaDB/MySQL在开启binlog的情况下单位时间内的事务提交数，就必须减少每个事务提交过程中导致的fsync的调用次数。MariaDB从版本5.3开始，引入了binlog group commit技术来解决这个问题，MySQL从版本5.6开始也加入了binlog group commit技术，其他一些非官方的组织也提交了自己的补丁来解决这个问题，但基本思路都非常相似。本章中，我们将详细讲解binlog group commit技术。

本章的内容主要包括：

- 事务的两阶段提交
- binlog group commit的工作原理
- binlog group commit的实现

7.1 事务的两阶段提交

MariaDB/MySQL在开启了binlog的情况下，因为MariaDB/MySQL是通过binlog进行复制的，为了保证数据在主库和从库之间的一致性，会使用事务的两阶段提交协议。同时，为了保证数据的安全性，我们还需要设置参数innodb_flush_logs_at_trx_commit = 1（如果我们使用的是InnoDB

事务存储引擎）以及参数`sync_binlog=1`，前者保证了事务在InnoDB存储引擎内的修改持久化到了磁盘（对于InnoDB来说是重做日志的持久化），后者保证了该事务在binlog中的修改持久化到了磁盘。

下面我们先来看一下事务在MariaDB/MySQL内部的两阶段提交过程，如图7-1所示。

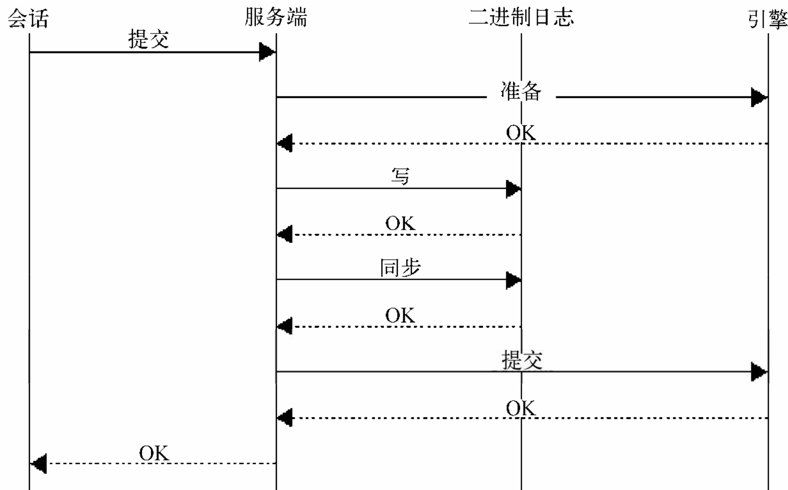


图7-1　事务的两阶段提交

从图7-1可以看出，事务的提交被划分为3个步骤。

(1) 事务在存储引擎内部准备好。处于准备好状态的事务可以被回滚，也可以被提交。

(2) 事务的所有修改写入到binlog中并进行持久化。这一步完成后，当数据库发生崩溃恢复的时候，存储引擎内部已经准备好但没有提交的事务可以通过binlog恢复。

(3) 事务在存储引擎内部提交。这一步完成后对应的binlog对于崩溃恢复来说就没有作用了。

事务的两阶段提交协议保证了无论在任何情况下，事务要么同时存在于存储引擎和binlog中，要么两个里面都不存在，这就保证了主库与从库之间数据的一致性。如果数据库系统发生崩溃，当数据库系统重新重启时会进行崩溃恢复操作，存储引擎中处于准备好状态的事务会去查询该事务是否同时也存在于binlog中，如果存在就在存储引擎内部提交该事务（因为此时从库可能已经获取了对应的binlog内容），如果binlog中没有该事务，就回滚该事务。例如，当崩溃发生在第一步和第二步之间时，明显处于准备好状态的事务还没来得及写入到binlog中，所以该事务会在存储引擎内部进行回滚，这样该事务在存储引擎和binlog中都不存在；当崩溃发生在第二步和第三步之间时，处于准备好状态的事务存在于binlog中，那么该事务会在存储引擎内部进行提交，这样该事务就同时存在于存储引擎和binlog中。

为了保证数据的安全性，以上列出的3个步骤都需要调用fsync将数据持久化到磁盘。由于在引擎内部准备好的事务可以通过binlog来恢复，所以通常情况下第三个fsync是可以省略的。

7.2 binlog group commit 的工作原理

group commit的核心思想是多个并发的需要提交的事务之间共享一个fsync操作来进行数据的持久化，将fsync操作的开销平摊到多个并发的事务上去。例如，有10个并发的事务需要提交，我们可以通过让这10个事务共享一次fsync操作进行持久化，这相比于每个事务需要自己执行一次fsync来进行持久化，性能上得到了明显提升。

由于开启binlog group commit没有任何性能方面或者其他方面的不良影响，所以默认情况下MariaDB的binlog group commit功能是开启的，不需要配置额外的参数来开启。同时我们需要注意的是，binlog group commit不是在任何时候都能发挥作用的，只有在有足够多并发的需要提交的事务时，fsync操作成为事务的提交瓶颈的情况下才能给MariaDB带来性能的提升。

MariaDB提供了binlog_commits和binlog_group_commits两个变量，让你能够看到binlog group commit技术为数据库系统减少了多少次fsync的调用。通过调用SHOW STATUS LIKE "binlog_%commits"命令，可以查看这两个变量的值。前者是在binlog中提交的总事务数，后者是执行的binlog group commit次数。这两个值会根据数据库系统压力的不同以及压力是持续的还是间断性的而千差万别。示例代码如下：

```
mysql> SHOW STATUS LIKE "binlog_%commits";
+--------------------------------+-------------+
| variable_name                  | Value       |
+--------------------------------+-------------+
| binlog_commits                 | 2836932     |
| binlog_group_commits           | 1815490     |
+--------------------------------+-------------+
2 rows in set (0.01 sec)
```

binlog_group_commits的值总是小于等于binlog_commits的值，因为一次group commit操作提交的事务数总是大于等于1的。

图7-2描述了binlog group commit技术中多个事务共享一次fsync操作对binlog进行持久化的工作原理，虚线部分为多个事务共享一次fsync操作对binlog文件进行同步。其实图中描述的和实际MariaDB/MySQL的binlog group commit实现还是有一些出入的，具体我们会在下一节中详细介绍，图7-2只是为了让读者对binlog group commit技术有个非常直观的理解。



图7-2　多个事务之间共享一次fsync操作进行binlog的持久化

7.3　binlog group commit 的实现

binlog group commit技术的具体实现如下：多个并发提交的事务在写binlog之前会被加入到一个队列中，位于队列头部的事务所在的线程称为leader线程，其他事务所在的线程称为follower线程。leader线程负责为队列中所有的事务进行写binlog操作，此时所有的follower线程处于等待状态，然后leader线程调用一次fsync操作，将binlog持久化，最后通知所有的follower线程可以继续往下执行。

接下来，我们从MariaDB源代码的角度来进一步了解binlog group commit技术的相关细节。

7.3.1　相关的数据结构

这里我们首先给出binlog group commit中用到的几个重要数据结构的定义以及相关说明。

1. binlog_cache_mngr类

binlog_cache_mngr的主要作用是缓存事务对binlog的修改，其定义如下：

```
// sql/loc.cc

class binlog_cache_mngr;
```

事务在提交（提交事务有3种方式：显示执行COMMIT命令；当前连接的autocommit设置为true，执行的每条语句对应单独的一个事务；某些DDL命令的执行会导致当前事务隐式地提交，比如

ALTER TABLE、CREATE TABLE等）之前，它对数据库的修改是不会直接写入到binlog中的，而是缓存在binlog_cache_mngr里，只有当事务提交时，才会将binlog_cache_mngr的所有内容写入到binlog中，当事务回滚时，直接丢弃binlog_cache_mngr中的所有内容。也就是说，写binlog时，会一次性写入一个完整的事务。例如，一个包含多条语句的事务，在执行COMMIT语句之前，所有对数据库的修改都被缓存在binlog_cache_mngr结构中，只有在执行COMMIT语句后，被缓存的内容才会被写入到binlog中。

2. group_commit_entry结构

接下来看一下结构group_commit_entry，这是binlog group commit中的核心数据结构，一个group_commit_entry实例对应即将写入binlog的一个事务：

```
// sql/log.h

struct group_commit_entry
{
    struct group_commit_entry *next;    // 实现队列
    THD *thd;                           // 事务的所有上下文
    binlog_cache_mngr *cache_mngr;
    bool using_stmt_cache;
    bool using_trx_cache;

    Log_event *end_event;               // 事务在binlog中的结束事件
    Log_event *incident_event;          /* 如果发生异常，将会在binlog中写入一个事故事件来告知从库、
                                           主库发生了异常，可能会导致主库和从库的数据不一致*/

    // 记录该事务在binlog group commit过程中发生的错误
    int error;
    int commit_errno;
    IO_CACHE *error_cache;

    bool all;                           // 为true表示一个事务结束

    bool need_unlog;                    // COMMIT检查点
    bool check_purge;                   // 为true表明binlog发生了切换

    bool queued_by_other;               // 为true表示被其他事务所在线程加入到组提交队列中
    ulong binlog_id;                    // 该事务写入到了哪个binlog中
};
```

下面给出了group_commit_entry中各个成员的具体含义。

- next：用于实现group_commit_entry队列。多个并发提交的事务在写binlog前会加入到一个队列中，由leader线程完成队列中所有事务的写binlog操作以及调用一次fsync对binlog进行持久化。

- ❑ thd：事务的所有上下文信息。
- ❑ cache_mngr：事务在提交之前，它对数据库的修改都会缓存在该结构内，等待事务提交时，全部写入到binlog中。
- ❑ using_stmt_cache：为true表示一个命令结束。
- ❑ using_trx_cache：为true表示一个事务结束。
- ❑ end_event：该事件在binlog中标识了当前事务的结束。
- ❑ incident_event：当主库上发生了异常时，会在binlog中记录一个事故事件，该事件用来通知从库，主库发生了异常，可能会导致主库和从库的数据不一致。
- ❑ error：记录组提交期间发生的错误（MariaDB自定义错误码）。
- ❑ commit_errno：记录组提交期间发生的错误（系统错误码）。
- ❑ error_cache：以文本的形式记录错误的具体信息。
- ❑ all：为true表示一个事务已经结束。
- ❑ need_unlog：如果为true，表示存储引擎不支持commit_checkpoint_request接口。commit_checkpoint_request接口是存储引擎提供给上层用于请求存储引擎内部建立检查。
- ❑ check_purge：如果为true，说明binlog发生了切换。
- ❑ queued_by_other：该事务被其他事务所在的线程加入到组提交队列中。
- ❑ binlog_id：标识了该事务写入到哪个binlog。

7.3.2　代码执行流程

当事务提交时，会调用ha_commit_trans函数，该函数负责：

- ❑ 事务在引擎内部准备好；
- ❑ 事务写入到binlog中；
- ❑ 在引擎内部提交该事务。

事务在引擎内部准备好以及在引擎内部提交事务不是我们关注的，这里重点关注事务写入到binlog中的过程。在ha_commit_trans函数内部，会调用类MYSQL_BIN_LOG的log_and_order函数完成对binlog的写入。MariaDB/MySQL对binlog的操作都是通过MYSQL_BIN_LOG类进行的。本节中，我们将把MYSQL_ BIN_LOG的log_and_order函数作为切入点来分析MariaDB的binlog group commit技术，相关的代码在sql/log.cc文件中：

```
// sql/log.cc

1. int MYSQL_BIN_LOG::log_and_order(THD *thd, my_xid xid, bool all,
       bool need_prepare_ordered __attribute__((unused)),
       bool need_commit_ordered __attribute__((unused)))
2. {
3.     int err;
```

```
4.     binlog_cache_mngr *cache_mngr= thd->binlog_setup_trx_data();
5.     if (!cache_mngr)
6.          DBUG_RETURN(0);

       // 写binlog
7.     cache_mngr->using_xa= TRUE;
8.     cache_mngr->xa_xid= xid;
9.     err= binlog_commit_flush_xid_caches(thd, cache_mngr, all, xid);

10.    if (err)
11.        DBUG_RETURN(0);

       // 将binlog_id信息放入返回值中
12.    if (!xid || !cache_mngr->need_unlog)
13         DBUG_RETURN(BINLOG_COOKIE_DUMMY(cache_mngr->delayed_error));
14.    else
15.        DBUG_RETURN(BINLOG_COOKIE_MAKE(cache_mngr->binlog_id,
16.            cache_mngr->delayed_error));
17.}
```

我们先来看一下函数log_and_order的几个参数的含义：thd存储了事务所在线程的所有上下文信息，其中也包括该事务的所有上下文信息；xid是该事务的id；all如果为true，表示这个事务要么是用户主动执行commit语句提交的或者是由于用户执行DDL语句被动提交的。函数的最后两个参数暂时没有使用。

第4行到第6行代码用于获取该事务相关的binlog_cache_mngr结构。我们之前提到过，该结构缓存了事务即将写入到binlog中的内容。真正地写binlog的操作发生在第9行，这里调用了类MYSQL_BIN_LOG的binlog_commit_flush_xid_caches函数。

第12行到第16行代码用于将该事务写入到哪个binlog文件这一信息放入返回值中。该信息在存储引擎不支持commit_checkpoint_request时使用。

继续往下分析，我们看一下函数binlog_commit_flush_xid_caches到底做了什么事情：

```
// sql/log.cc

1. static inline int binlog_commit_flush_xid_caches(THD *thd,
       binlog_cache_mngr *cache_mngr, bool all, my_xid xid)
2. {
3.     if (xid)
4.     {
           // 以一个xid事件作为事务的结束
5.         Xid_log_event end_evt(thd, xid, TRUE);
6.         return (binlog_flush_cache(thd, cache_mngr, &end_evt, all, TRUE, TRUE));
7.     }
8.     else
```

```
9.     {
           /*
             当xid为空时，我们以一个内容为COMMIT的QUERY_EVENT事件来标记该事务结束
           */
10.        Query_log_event end_evt(thd, STRING_WITH_LEN("COMMIT"),TRUE, TRUE, TRUE, 0);
11.        return (binlog_flush_cache(thd, cache_mngr, &end_evt, all, TRUE, TRUE));
12.    }
13.}
```

在上面这个函数中，我们看到Xid_log_evnet和Query_log_event这两个事件类型。第6章已介绍过，binlog以事件的形式记录数据库的变更情况，所有在代码中看到的是以_log_event结尾的不同类型代表了binlog的不同事件。

第3行到第7行代码中，如果xid不为空，在binlog中以一个xid事件标记一个事务的结束。

第8行到第12行代码中，当xid为空时，我们以一个内容为COMMIT的QUERY_EVENT事件（详见6.3节）来标识该事务的结束。

不管代码走哪个分支，最后都会调用binlog_flush_cache函数，将事务对数据库做的更改写入到binlog中。下面我们接着分析一下binlog_flush_cache函数：

```
// sql/log.cc

1. static int binlog_flush_cache(THD *thd, binlog_cache_mngr *cache_mngr,Log_event *end_ev, bool all,
       bool using_stmt, bool using_trx)
2. {
3.     int error= 0;

4.     if ((using_stmt && !cache_mngr->stmt_cache.empty()) ||
5.         (using_trx && !cache_mngr->trx_cache.empty()))
6.     {
           // 将所有没有写入到binlog_cache_mngr的事件写入到binlog_cache_mngr中
7.         if (using_stmt && thd->binlog_flush_pending_rows_event(TRUE, FALSE))
8.             DBUG_RETURN(1);
9.         if (using_trx && thd->binlog_flush_pending_rows_event(TRUE, TRUE))
10.            DBUG_RETURN(1);

           // 将事务的所有事件写入到binlog中
11.        error= mysql_bin_log.write_transaction_to_binlog(thd, cache_mngr,
               end_ev, all, using_stmt, using_trx);
12.    }
13.    else
14.    {
15.        cache_mngr->need_unlog= 0;
16.    }

       // 清理binlog_cache_mngr
17.    cache_mngr->reset(using_stmt, using_trx);
```

```
18.    DBUG_ASSERT((!using_stmt || cache_mngr->stmt_cache.empty()) &&
           (!using_trx || cache_mngr->trx_cache.empty()));
19.    DBUG_RETURN(error);
20.}
```

从代码中可以看出，该函数的主要功能就是在事务提交时，将还未写入到binlog_cache_mngr的内容全部写入到binlog_cache_mngr中（第7行到第10行代码），然后调用MYSQL_BIN_LOG的write_transaction_to_binlog函数将该事务产生的所有修改写入到binlog中（第11行代码），接着清空binlog_cache_mngr结构中的内容，以便开始新的事务（第17行代码）。接下来，让我们跟随代码流程进入write_transaction_to_binlog函数：

```
1. bool MYSQL_BIN_LOG::write_transaction_to_binlog(THD *thd,
                                                   binlog_cache_mngr *cache_mngr,
                                                   Log_event *end_ev, bool all,
                                                   bool using_stmt_cache,
                                                   bool using_trx_cache)
2. {
3.     group_commit_entry entry;
4.     Ha_trx_info *ha_info;

       // 给entry结构赋值
5.     entry.thd= thd;
6.     entry.cache_mngr= cache_mngr;
7.     entry.error= 0;
8.     entry.all= all;
9.     entry.using_stmt_cache= using_stmt_cache;
10.    entry.using_trx_cache= using_trx_cache;
11.    entry.need_unlog= false;

12.    ha_info= all ? thd->transaction.all.ha_list : thd->transaction.stmt.ha_list;
13.    for (; ha_info; ha_info= ha_info->next())
14.    {
15.        if (ha_info->is_started() && ha_info->ht() != binlog_hton &&
16.            !ha_info->ht()->commit_checkpoint_request)
17.            entry.need_unlog= true;
18.        break;
19.    }

20.    entry.end_event= end_ev;

       // 如果在执行命令或者事务的过程中发生了异常，记录事故事件
21.    if (cache_mngr->stmt_cache.has_incident() ||
           cache_mngr->trx_cache.has_incident())
22.    {
23.        Incident_log_event inc_ev(thd, INCIDENT_LOST_EVENTS, write_error_msg);
24.        entry.incident_event= &inc_ev;
25.        DBUG_RETURN(write_transaction_to_binlog_events(&entry));
```

```
26.     }
27.     else
28.     {
29.         entry.incident_event= NULL;
30.         DBUG_RETURN(write_transaction_to_binlog_events(&entry));
31.     }
32.}
```

从以上代码可以看出，write_transaction_to_binlog函数首先将事务的内容封装在group_commit_entry结构里，然后调用write_transaction_to_binlog_events函数将entry的内容写入到binlog中。

第5行到第11行代码用于填充结构group_commit_entry的值。

第12行到第19行代码中，如果存储引擎不支持commit_checkpoint_request接口，那么将entry的need_unlog赋值成true。

第21行到第26行代码中，如果在执行事务的过程中发生了异常，会记录一个incident事件到binlog，用来通知从库，主库发生了异常，可能会导致主库和从库数据的不一致。

下面我们接着分析write_transaction_to_binlog_events函数的工作：

```
1. bool MYSQL_BIN_LOG::write_transaction_to_binlog_events(group_commit_entry *entry)
2. {
       // 将待提交到binlog的事务加入到组提交队列中
3.     bool is_leader= queue_for_group_commit(entry);

4.     if (is_leader)  // leader线程
5.         trx_group_commit_leader(entry);
6.     else if (!entry->queued_by_other)  // follower线程
7.         entry->thd->wait_for_wakeup_ready();
8.     else
9.     {
           /*
             entry->queued_by_other为true，表示该事务被其他事务所在线程加入到了队列中，
             执行到这里，说明leader已经完成了所有工作，并且唤醒了当前线程
           */
10.    }

       // 没有错误，直接返回
11.    if (likely(!entry->error))
12.        return 0;

       // 相关的错误处理
13.    ...

14.    return 1;
15.}
```

第3行代码调用queue_for_group_commit函数将待提交到binlog的事务加入到组提交队列中。

第4行到第5行代码中，我们将组提交队列中第一个事务所在的线程作为leader线程调用trx_group_commit_leader函数，负责将组提交队列中的所有事务提交，然后调用一次fsync对binlog进行持久化，最后唤醒所有follower线程继续往下执行。

第6行到第7行代码中，所有的follower线程进入等待队列，直到leader线程在完成工作后将其唤醒。

第8行到第10行代码中，如果entry->queued_by_other为true，说明该事务已经被其他事务所在的线程加入到了组提交队列中。发生这种情况的原因是事务之间的提交顺序有依赖关系，这在下一节中会详细介绍。

7.3.3 事务排队

接下来，我们看一下待提交的事务是如何在组提交队列中排队的。

下面列出了MYSQL_BIN_LOG类的queue_for_group_commit函数的代码，该函数的主要功能是将事务加入到组提交队列中：

```
// sql/log.cc

1. bool MYSQL_BIN_LOG::queue_for_group_commit(group_commit_entry *orig_entry)
2. {
3.     group_commit_entry *entry, *orig_queue;
4.     wait_for_commit *list, *cur, *last;
5.     wait_for_commit *wfc;

6.     wfc= orig_entry->thd->wait_for_commit_ptr;
7.     orig_entry->queued_by_other= false;

       // 是否需要等待其他事务提交完毕后才提交
8.     if (wfc && wfc->waiting_for_commit)
9.     {
           /*
             在有锁的情况下二次检查waiting_for_commit的值，
             避免在我们等待之前waiting_for_commit的值被改变
           */
10.        mysql_mutex_lock(&wfc->LOCK_wait_commit);
11.        if (wfc->waiting_for_commit)
12.        {
               /*
                 等待其他事务提交完毕。被等待的事务会将当前事务加入到
                 组提交队列中，那么当前事务的queue_by_other将会是true
               */
```

```
13.             wfc->opaque_pointer= orig_entry;
14.             do
15.             {
16.                 mysql_cond_wait(&wfc->COND_wait_commit,
17.                     &wfc->LOCK_wait_commit);
18.             } while (wfc->waiting_for_commit);
19.                 wfc->opaque_pointer= NULL;
20.         }
21.         mysql_mutex_unlock(&wfc->LOCK_wait_commit);
22.     }

        // 如果当前事务已经被等待提交的事务加入到组提交队列中，则直接返回
23.     if (orig_entry->queued_by_other)
24.         DBUG_RETURN(false);

        // 获取组提交队列，并且获取锁保护
25.     orig_entry->thd->clear_wakeup_ready();
26.     mysql_mutex_lock(&LOCK_prepare_ordered);
27.     orig_queue= group_commit_queue;

28.     list= wfc;
29.     cur= list;
30.     last= list;
31.     entry= orig_entry;
32.     for (;;)
33.     {
            // 将事务加入到组提交队列
34.         entry->next= group_commit_queue;
35.         group_commit_queue= entry;

36.         if (entry->cache_mngr->using_xa)
37.         {
38.             run_prepare_ordered(entry->thd, entry->all);
39.         }

            // 如果没有事务需要等待orig_entry提交完成后才能提交，cur为NULL
40.         if (!cur)
41.             break;

            /*
              接下来处理等待在cur上的所有事务。如果有事务A需要等待cur提交后才能提交，同时事务B需要
              等待事务A提交后才能提交，需要将这些准备递归地提交到binlog的事务加入到队列中
            */
42.         if (cur->subsequent_commits_list)
43.         {
44.             bool have_lock;
45.             wait_for_commit *waiter;

46.             mysql_mutex_lock(&cur->LOCK_wait_commit);
```

```
47.             have_lock= true;

                // 处理所有等待在cur上的事务
48.             waiter= cur->subsequent_commits_list;
49.             cur->subsequent_commits_list= NULL;
50.             while (waiter)
51.             {
52.                 wait_for_commit *next= waiter->next_subsequent_commit;
53.                 group_commit_entry *entry2=
54.                     (group_commit_entry *)waiter->opaque_pointer;
55.                 if (entry2)
56.                 {
                        /*
                          该事务（waiter）准备提交到binlog中，将其加入到链表的尾部，
                          这样外部的循环就能将其加入组提交队列，
                          并且递归处理等待在该事务上的其他事务
                        */
57.                     entry2->queued_by_other= true;
58.                     last->next_subsequent_commit= waiter;
59.                     last= waiter;
60.             }
61.             else
62.             {
                    // 唤醒等待的事务
63.                 if (have_lock)
64.                 {
65.                     have_lock= false;
66.                     cur->wakeup_subsequent_commits_running= true;
67.                     mysql_mutex_unlock(&cur->LOCK_wait_commit);
68.                 }
69.                 waiter->wakeup(0);
70.             }
71.             waiter= next;
72.         }
73.         if (have_lock)
74.             mysql_mutex_unlock(&cur->LOCK_wait_commit);
75.     }

        // 查看链表里的事务是否全部处理完
76.     if (cur == last)
77.         break;

        // 处理链表中的下一个需要加入到组提交队列的事务
78.     cur= cur->next_subsequent_commit;
79.     entry= (group_commit_entry *)cur->opaque_pointer;
80.     DBUG_ASSERT(entry != NULL);
81. }
```

```
82.    if (opt_binlog_commit_wait_count > 0)
83.        mysql_cond_signal(&COND_prepare_ordered);
84.    mysql_mutex_unlock(&LOCK_prepare_ordered);

      /*
        如果orig_queue为空，说明当前事务是组提交队列中的第一个事务，
        那么当前线程将成为leader线程
      */
85.    DBUG_RETURN(orig_queue == NULL);
86.}
```

第8行到第22行代码说明，如果该事务需要等待其他事务提交之后才能提交，那么在当前事务加入到组提交队列之前，必须做相应的等待。第13行到第19行代码说明进入等待状态。

第23行和第24行代码说明，如果orig_entry->queued_by_other为true表示当前事务已经被其他事务所在的线程加入到组提交队列中。这种情况发生在当前事务需要等待其他事务提交后才能提交，那么当前事务所在的线程会进入等待状态，被等待事务所在的线程在将自己的事务加入到组提交队列之后，也会将所有等待在它上面的其他事务按顺序加入到队列中，当leader线程将组提交队列中的事务按顺序写入到binlog中并且调用存储引擎的commit_ordered后，会唤醒所有等待的follower线程。

第26行代码获取全局锁LOCK_prepare_ordered，该锁在这里是为了保护组提交队列，防止多个线程同时对其进行操作。获取了锁保护之后，我们就可以将事务添加到组提交队列中。

第32行到第81行代码的主要工作就是将事务添加到组提交队列中，其中处理了其他事务需要等待当前事务提交后才能提交。例如，事务B需要等待事务A提交后才能提交，同时事务C需要等待事务B提交后才能提交，那么这三个事务进入组提交队列的顺序应该是A、B、C。代码中的两个循环就是为了处理这种递归依赖的事务之间入队顺序的问题。

第82行和第83行代码处理用户设置了binlog_group_commit_wait_count参数的情况。当用户设置了该参数，需要等待组提交队列中至少有该参数指定个数的事务时才进行group commit操作，否则leader线程进入等待。这里是通知leader线程，当有新的事务加入到组提交队列中时，则让leader线程去检查是否满足进行组提交的条件。

第85行代码说明如果当前事务是组提交队列中的第一个事务，那么当前事务所在的线程将作为leader线程执行组提交的后续操作。

7.3.4　leader线程的工作

前面介绍过，组提交队列中第一个事务所在的线程将以leader线程的身份执行队列中所有事务的写binlog操作以及binlog的fsync操作，最后将所有等待的follower线程唤醒。

下面给出了leader线程的执行函数trx_group_commit_leader的代码，我们通过分析该函数的

代码来了解leader线程的具体工作：

```
// sql/log.cc

1. void MYSQL_BIN_LOG::trx_group_commit_leader(group_commit_entry *leader)
2. {
3.     uint xid_count= 0;
4.     my_off_t UNINIT_VAR(commit_offset);
5.     group_commit_entry *current, *last_in_queue;
6.     group_commit_entry *queue= NULL;
7.     bool check_purge= false;
8.     ulong binlog_id;
9.     uint64 commit_id;
10.    binlog_id= 0;

11.    {
         /*
           由于需要对binlog进行写操作，首先获取LOCK_log锁保护。
           对组提交队列的操作需要用到LOCK_prepare_ordered锁保护
         */
12.        mysql_mutex_lock(&LOCK_log);
13.        mysql_mutex_lock(&LOCK_prepare_ordered);

         /*
           如果配置了binlog_commit_wait_cout参数，组提交队列需要
           收集一定数量的事务才能执行group commit操作。
           wait_for_sufficient_commits()内部会释放并且重新获取
           LOCK_log和LOCK_prepare_ordered
         */
14.        if (opt_binlog_commit_wait_count)
15.            wait_for_sufficient_commits();

          /*
            接下来将group_commit_queue的内容放到current里面，
            这就是group commit本次将要处理的事务集合，
            新的事务在LOCK_prepare_ordered锁释放之后
            可以继续添加到group_commit_queue中
          */
16.        current= group_commit_queue;
17.        group_commit_queue= NULL;
18.        mysql_mutex_unlock(&LOCK_prepare_ordered);
19.        binlog_id= current_binlog_id;

          // 由于队列中的元素是逆序的，需要进行反转
20.        last_in_queue= current;
21.        while (current)
22.        {
23.            group_commit_entry *next= current->next;
24.            current->next= queue;
```

7

```
25.             queue= current;
26.             current= next;
27.         }
28.         DBUG_ASSERT(leader == queue);       // leader应该是queue的第一个元素
29.     }

30.     if (likely(is_open()))       // 应该一直为true
31.     {
32.         commit_id= (last_in_queue == leader ? 0 : (uint64)leader->thd->query_id);

        /*
          将队列中的事务提交到binlog中。

          需要注意的是，由于所有的操作都是由leader线程完成的，
          所以需要保存每个事务提交过程中的错误码，
          等待事务所在的线程被leader线程唤醒后对其进行检查，然后执行相应的操作
        */
33.         for (current= queue; current != NULL; current= current->next)
34.         {
35.             binlog_cache_mngr *cache_mngr= current->cache_mngr;

36.             if ((current->error= write_transaction_or_stmt(current, commit_id)))
37.                 current->commit_errno= errno;

38.             strmake_buf(cache_mngr->last_commit_pos_file, log_file_name);
39.             commit_offset= my_b_write_tell(&log_file);
40.             cache_mngr->last_commit_pos_offset= commit_offset;
41.             if (cache_mngr->using_xa && cache_mngr->xa_xid)
42.             {
                /*
                  如果存储引擎不支持commit_checkpoint_request，往binlog文件中
                  写入事务时，需要增加该binlog文件对应的计数器计数
                */
43.                 if (current->need_unlog)
44.                 {
45.                     xid_count++;
46.                     cache_mngr->need_unlog= true;
47.                     cache_mngr->binlog_id= binlog_id;
48.                 }
49.                 else
50.                     cache_mngr->need_unlog= false;

51.                 cache_mngr->delayed_error= false;
52.             }
53.         }

        // 调用一次fsync对binlog进行持久化
54.         bool synced= 0;
55.         if (flush_and_sync(&synced))
```

```
56.         {
                // 如果发生错误，保存错误码
57.             for (current= queue; current != NULL; current= current->next)
58.             {
59.                 if (!current->error)
60.                 {
61.                     current->error= ER_ERROR_ON_WRITE;
62.                     current->commit_errno= errno;
63.                     current->error_cache= NULL;
64.                 }
65.             }
66.         }
67.         else
68.         {
69.             bool any_error= false;
70.             bool all_error= true;
71.             for (current= queue; current != NULL; current= current->next)
72.             {
73.                 if (!current->error &&
                        RUN_HOOK(binlog_storage, after_flush,
                        (current->thd, log_file_name,
                        current->cache_mngr->last_commit_pos_offset, synced)))
74.                 {
75.                     current->error= ER_ERROR_ON_WRITE;
76.                     current->commit_errno= -1;
77.                     current->error_cache= NULL;
78.                     any_error= true;
79.                 }
80.                 else
81.                     all_error= false;
82.             }
83.
84.             if (any_error)
85.                 sql_print_error("Failed to run 'after_flush' hooks");

                // binlog有更新，通知潜在的等待者（例如复制中的dump线程）
86.             if (!all_error)
87.                 signal_update();
88.         }

            // 增加该binlog文件对应的计数器计数
89.         if (xid_count > 0)
90.         {
91.             mark_xids_active(binlog_id, xid_count);
92.         }

            // 检查binlog的大小，决定是否进行切换
93.         if (rotate(false, &check_purge))
94.         {
```

7

```
95.             leader->cache_mngr->delayed_error= true;
96.             my_error(ER_ERROR_ON_WRITE, MYF(ME_NOREFRESH),name, errno);
97.             check_purge= false;
98.         }
99.     }

100.    mysql_mutex_lock(&LOCK_commit_ordered);
101.    last_commit_pos_offset= commit_offset;

        /*
          在获取LOCK_commit_ordered锁之前不能释放LOCK_log锁，
          否则下一个组提交操作可能在我们之前执行commit_ordered，
          导致事务在引擎内部的提交顺序与binlog中的顺序不一致
        */
102.    mysql_mutex_unlock(&LOCK_log);

        // 统计组提交操作的执行次数
103.    ++num_group_commits;

        // 在引擎内部按顺序执行commit_ordered操作，然后唤醒所有follower线程
104.    current= queue;
105.    while (current != NULL)
106.    {
107.        group_commit_entry *next;

            // 统计提交了多少个事务
108.        ++num_commits;

            /*
              对于事务的两阶段提交协议，为了保证事务在存储引擎内部的提交顺序
              和binlog中一致，需要根据事务在组提交队列中的顺序调用存储
              引擎的commit_ordered接口
            */
109.        if (current->cache_mngr->using_xa && !current->error &&
                DBUG_EVALUATE_IF("skip_commit_ordered", 0, 1))
110.            run_commit_ordered(current->thd, current->all);

            /*
              在唤醒current所在线程之前，取出current->next的值，
              因为current所在的线程被唤醒后可能会修改next的值
            */
111.        next= current->next;
112.        if (current != leader)                      // leader线程自己不需要唤醒
113.        {
114.            if (current->queued_by_other)
115.                current->thd->wait_for_commit_ptr->wakeup(current->error);
116.            else
117.                current->thd->signal_wakeup_ready();
118.        }
```

```
119.        current= next;
120.    }
121.    mysql_mutex_unlock(&LOCK_commit_ordered);

        /*
          如果check_purge为true，说明binlog发生了切换，
          请求存储引擎持久化所有提交的事务，加快崩溃恢复的速度
        */
122.    if (check_purge)
123.        checkpoint_and_purge(binlog_id);

124.    DBUG_VOID_RETURN;
125.}
```

第12行代码获取了锁LOCK_log，该锁用于对binlog进行保护，因为后续我们会对binlog进行写操作。

第13行代码获取了锁LOCK_prepare_ordered，这是一个全局锁，它的作用是保护组提交队列。

第14行和第15行代码说明，如果配置了MariaDB的binlog_commit_wait_count参数，会检查组提交队列中的总事务数是否大于等于该参数指定的值，如果小于该值，就需要等待，直到队列中的事务数大于等于该值，或者等待的时间超过了参数binlog_commit_wait_usec指定的值。参数binlog_commit_wait_count的默认值是0，所以默认情况下，即使组提交队列中只有一个事务也不会等待。wait_for_sufficient_commits函数内部会释放并且重新获取LOCK_log和LOCK_prepare_ordered。

第16行和第17行代码是两次简单的指针赋值操作，将组提交队列中的事务放入current队列中，并且清空组提交队列，本次group commit的后续所有操作都是针对current队列来进行的。

第18行代码用于释放锁LOCK_prepare_ordered，那么新的事务可以加入到group commit队列中来，新加入的事务会是下一次binlog group commit操作的对象。我们可以看到，这里LOCK_prepare_ordered锁持有的时间是非常短暂的，不会影响新的事务添加到组提交队列中来。

第30行代码用于判断binlog是否处于开启状态。

第33到第53行代码的主要工作是将队列中的事务按顺序写入binlog中的IO_CACHE缓存中。

第55行代码用于将IO_CACHE缓存中的内容刷新到binlog文件中，并且调用一次fsync操作对binlog进行持久化。在此之后，如果系统发生崩溃，事务可以根据binlog的内容进行恢复。

第93行代码中，由于我们对binlog进行了写入操作，所以需要检查当前binlog文件的大小，如果它大于等于max_binlog_size设定的值，binlog需要进行切换操作，新建binlog文件以记录新的事务。

第100行到第121行代码的主要工作是按照事务在binlog中的顺序调用存储引擎的commit_ordered接口（该接口可以保证事务在存储引擎内部的提交顺序），然后唤醒所有的follower

线程。第100行到第102行首先获取了锁LOCK_commit_ordered，然后释放锁LOCK_log，这两个操作的顺序是不能颠倒的，如果颠倒的话，下一个group commit操作可能跑到本次group commit操作的前面获取LOCK_commit_ordered锁，并且执行存储引擎的commit_ordered接口，此时事务在存储引擎内部的提交顺序就和binlog中的顺序不一致了，这会导致数据在主库和从库的不一致。

第103行代码用于统计group commit操作的执行次数，对应于前面提到的binlog_group_ commits变量。第108行代码用于统计提交的事务数，对应于MariaDB的binlog_commits变量。第109行和第110行代码说明，如果采用的是事务的两阶段提交协议，就按照事务写入binlog的顺序调用存储引擎的commit_ordered函数（该函数在7.3.5节中介绍），这样保证了存储引擎内部的提交顺序和binlog中的顺序一致。第112行到第118行代码用于唤醒等待的follower线程，让它们继续执行。

第122行和第123行代码说明，如果binlog发生了切换，请求存储引擎建立检查点。我们之前介绍过，在MariaDB中事务的两阶段提交分为3个步骤：事务在引擎内部准备好，事务提交到binlog中以及事务在引擎内部提交。为了数据的安全性，每一步都需要调用fsync将数据持久化。由于在存储引擎内部已经准备好的事务可以根据binlog恢复，所以通常情况下第三步的fsync操作省略了。同时，为了加快崩溃恢复的速度，MariaDB会在binlog发生切换操作时，请求存储引擎建立检查点，这样在崩溃恢复时，MariaDB只需要扫描最新的一个binlog就可以了。

7.3.5　prepare_ordered和commit_ordered接口

在组提交的过程中，有可以并行执行的部分，也有需要串行执行的部分。为了让存储引擎和binlog方便地参与并行和串行的部分，MariaDB为存储引擎添加了两个接口：prepare_ordered和commit_ordered。添加这两个接口的目的是让它们执行group commit的串行部分，而原有的prepare和commit接口执行group commit的并行部分。

prepare_ordered函数在prepare函数执行后调用。在一个引擎内部，同一时刻只能有一个事务调用该函数，也就是说该函数必须串行调用。调用prepare_ordered函数，保证了事务在存储引擎和binlog中的准备顺序一致。commit_ordered函数在commit函数调用前调用。和prepare_ordered类似，调用commit_ordered函数保证了事务在所有存储引擎和binlog中的提交顺序一致。

由于prepare_ordered和commit_ordered必须串行执行，所以在这两个函数内部应该做尽可能少的事情，将慢速的工作放入prepare和commit中进行，保证这两个函数能够快速执行。在commit_ordered函数返回后，事务在引擎内被标记为已提交，同时调用commit_ordered的顺序也是事务在引擎内部的提交顺序（例如，在InnoDB中，commit_ordered函数会将事务的提交顺序记录到事务日志的缓冲区中）。

prepare_ordered和commit_ordered只会在事务使用两阶段提交协议的时候被调用。对于MariaDB/MySQL来说，如果关闭了binlog，由于内部采取的是一阶段事务提交协议，所以这种情况下这两个函数是不会被调用的。

prepare_ordered和commit_ordered这两个接口是可选的，存储引擎可以选择性地实现这两个接口。如果存储引擎选择不实现某个接口，那么对于该存储引擎来说，在group commit的相应阶段就没有了顺序的保证。例如，InnoDB在准备阶段不需要保证顺序，所以InnoDB没有实现prepare_ordered函数，但是实现了commit_ordered函数，这可以保证事务在binlog中的顺序和InnoDB内部的提交顺序一致。

7.4 小结

本章中，我们详细分析了MariaDB的binlog group commit技术。binlog group commit的基本思想是多个待提交的事务之间共享一次fsync操作对binlog进行持久化，这样就将fsync的开销平摊到了多个事务上。

binlog group commit技术在具有大量并发待提交事务的场景下能够提高单位时间内事务的提交数，从而提高系统的整体性能。

第 8 章 复制

MariaDB/MySQL的复制功能允许你从一台服务器将数据复制到另一台服务器上，让两台服务器的数据保持同步。这有利于我们构造高可用的应用，也为高性能、可扩展性、备份、灾难恢复等工作提供了可行的方案。从版本5.5开始，MariaDB/MySQL以插件的形式支持半同步复制。MySQL的延迟复制功能可以指定从库的数据相对于主库的数据有一定的延时，利用这一特性可以很容易实现数据库的回滚功能。MariaDB 10.0引入了多源复制，允许一个从数据库同步多个数据源的数据。在本章中，我们会对这些内容进行详细的讲解。

本章的内容主要包括：

- 简介
- 复制的作用
- 复制的工作原理
- 复制的配置
- 复制的实现
- 半同步复制
- 并行复制
- 多源复制
- GTID

8.1 简介

MariaDB/MySQL的复制功能支持多种不同结构的主从配置，包括一主一从、一主多从、从库同时又可以是其他库的主库，等等。MariaDB在10.0版本中还引入了多源复制功能，可以为一个从库配置多个数据源，这样从库就拥有多个主库的数据。多源复制可以很方便地把不同机器上的数据聚集到一起进行分析或者备份等工作。

MariaDB/MySQL的复制功能是基于binlog的，binlog记录了数据库的库表结构变更以及表数据的所有修改情况。从库获取主库的binlog内容，然后在本地重放，达到拥有和主库相同数据的

目的。根据主库设置的binlog格式的不同，复制分为基于行的复制（binlog格式为ROW）和基于语句的复制（binlog格式为STATEMENT）。基于语句的复制是在MySQL 3.23中引入的，而基于行的复制是MySQL 5.1版本才引入的。由于ROW格式的binlog产生的binlog文件比STATEMENT格式大，所以基于行的复制产生的网络流量较高，这在网络状况不是很好的环境下可能会使主库和从库之间产生较大的延时，但ROW格式的binlog具有幂等性，所以基于行的复制被认为是最安全的。不管是基于语句的复制还是基于行的复制，都推荐在网络环境比较好的情况下使用复制功能，这样就能保证从库和主库之间的数据具有较低的延时。

启用复制功能并不会增加服务器太多的开销，主要就是开启binlog带来的开销（binlog文件的追加写操作开销。如果配置了sync_binlog=1参数，还包括系统调用fsync带来的开销）。相对于复制带来的好处，这么少的开销还是值得的。

MariaDB/MySQL的复制功能具有向后兼容性。由于较新版本的MariaDB/MySQL的binlog中可能会引入新的事件类型，因此以较新版本的MariaDB/MySQL作为从库的时候可以识别并处理老版本MariaDB/MySQL的所有binlog事件。相反，如果以较新版本的MariaDB/MySQL作为主库，而较老版本的MariaDB/MySQL作为从库，binlog中可能包含不能识别的事件类型从而引发错误。推荐在使用复制功能时，保持主库和从库使用同样版本的MariaDB/MySQL，这样就不会由于版本不一致而产生问题。

MariaDB/MySQL的复制功能是单向的，是异步进行的。MariaDB/MySQL从5.5开始，以插件形式引入了半同步复制功能。当主库提交一个事务之后，只有当该事务传播到至少一个从库上并且写入到磁盘中之后，该事务在主库上才返回给客户端。半同步复制能够最大限度地保证主库和从库之间数据的一致性，但降低了系统的整体性能。在本章后续内容中，我们将详细介绍半同步复制功能。

MySQL从5.6开始支持延迟复制功能，通过配置延迟复制，能够保证从库的数据比主库的数据延迟指定的时间。你可以通过MySQL的延时复制功能来实现数据库回滚功能。

8.2 复制的作用

复制功能的一些常见用途如下所示。

- **水平扩展**。配置多个从库，将所有的读请求都在从库上处理，主库只负责写操作，这能在一定程度上提高主库的写效率。此外，也可以通过增加从库的个数来达到提高读请求的吞吐率的目地。这在读密集型的场景中很有价值。
- **数据备份**。由于从库复制了主库的所有内容，同时复制过程可以随时停止，并且可以重新开启，这样我们就可以停止从库的复制工作，对从库进行备份工作，然后重新开启复制工作。这样就能在不影响主库性能的情况下实现数据的备份。

- **数据分析**。如果直接在主库上进行数据分析等操作，必然会对主库的性能产生严重的影响。有了复制功能，我们可以将数据分析工作应用到从库上，数据分析结束之后重新开启复制功能。
- **数据分布**。当位于北京总部的同事想使用你位于南京数据节点上的数据，而你又不想他们直接访问你的数据库时，可以通过配置复制功能在北京的数据节点上配置一个从节点，将主库的数据复制到从库上，让他们直接访问从库。
- **高可用性**。复制功能能够解决单点故障问题。关掉主库之后将业务切换到从库上，从库继续提供服务，这对构造高可用的服务提供了有力的支撑。

8.3　复制的工作原理

本节中，我们概要讲解复制的基本工作原理，接着介绍中断日志（relay-log）的作用以及master.info文件和relay-log.info文件的作用。

8.3.1　概要

MariaDB/MySQL的复制功能是基于binlog进行的。每个从库都会建立一个到主库的连接，接收主库的binlog内容，然后在本地重放，使从库和主库之间的数据达到同步的状态。每个从库在本地都记录了当前的复制进度以及连接到主库所需要的信息等内容，所以你可以随时调用`STOP SLAVE`命令停止从库的复制工作，然后在完成备份或者数据分析等工作之后，再使用`START SLAVE`命令重新开启复制功能。

MariaDB/MySQL的复制工作主要由主库上的master dump线程、从库上的slave IO线程以及slave SQL线程来完成的。

当从库连接到主库上时，主库会创建一个master dump线程，该线程负责将binlog的内容发送给从库。你可以在主库上执行`SHOW PROCESSLIST`命令来查看到该线程：

```
mysql> SHOW PROCESSLIST;
+-------+-------+-------------------+-------+---------------+------+----------------------+
| Id    | User  | Host              | ...   | Command       | Time | State                |
+-------+-------+-------------------+-------+---------------+------+----------------------+
| 1     | root  | localhost:56608   | ...   | Query         | 0    | init                 |
| 4     | slave | localhost:56642   | ...   | Binlog Dump   | 11   | Master has sent...   |
+-------+-------+-------------------+-------+---------------+------+----------------------+
```

当在从库上执行`START SLAVE`语句来开启复制功能时，会创建一个slave IO线程和一个slave SQL线程。slave IO线程负责连接到主库，然后接收主库master dump线程发送过来的binlog内容，写到本地的relay-log中。slave SQL线程负责重放relay-log中的内容，将主库的所有修改反映到从库上。

在从库上执行SHOW PROCESSLIST或者SHOW SLAVE STATUS命令，可以查看slave IO线程和slave SQL线程是否在运行，相关代码如下：

```
mysql> SHOW SLAVE STATUS \G;
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 127.0.0.1
                  Master_User: slave
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000001
          Read_Master_Log_Pos: 151
               Relay_Log_File: slave-relay-log.000002
                Relay_Log_Pos: 361
        Relay_Master_Log_File: mysql-bin.000001
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
...
```

SHOW SLAVE STATUS命令除了能查看slave IO线程和slave SQL线程的运行状态外，还可以查看复制的进度信息。从上面的输出信息可以看出，slave IO线程已经接收到了主库mysql-bin.000001文件的偏移量为151的地方，重放过程已经执行到了本地slave-relay-log.000002文件的偏移量为361的地方。

复制的大概工作过程可以简单总结为如下3步，如图8-1所示。

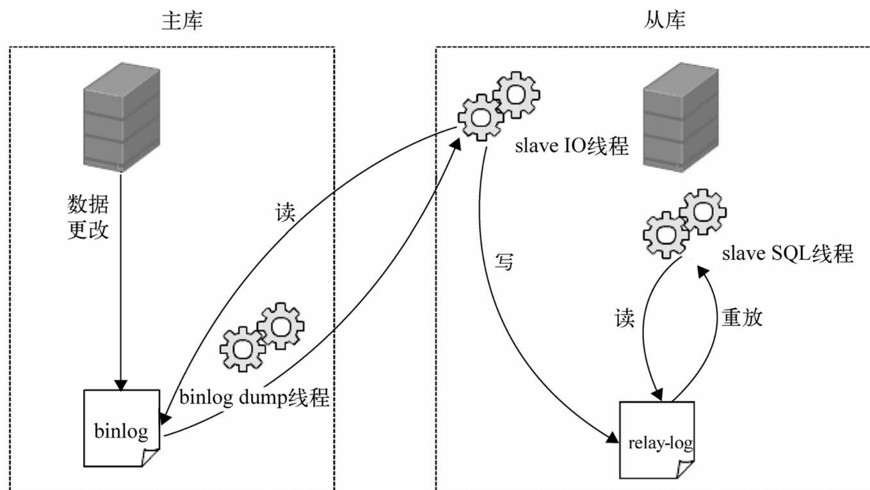


图8-1　复制原理图

(1) 主库将所有的修改以事件的形式记录到binlog中，主库的master dump线程负责发送binlog内容到从库。

(2) 从库的slave IO线程将接收到的binlog事件记录到本地的relay-log中。
(3) 从库的slave SQL线程重放relay-log中的事件。

在MySQL 4.0之前，复制是通过两个线程而不是3个线程完成的。在从库端，只有一个线程同时负责接收主库发来的binlog事件以及事件的重放工作，所以没有使用relay-log。这样做的缺点就是，当binlog事件重放速度较慢时，会直接影响读取binlog事件的进行。将事件的读取和重放工作分别放在不同的线程里来做，实现了事件获取和事件重放的解耦，这两个动作是完全异步执行的。

8.3.2　relay-log

在复制的过程中，从库的slave IO线程获取主库的binlog事件后，将其写入到本地的relay-log中，而slave SQL线程从relay-log读取事件并进行重放。relay-log.info文件记录了slave SQL线程重放的进度等信息，保证了停止复制之后再重新开启复制时，复制工作能够从正确的位置开始。请不要手动修改relay-log.info文件的内容，否则将会出现不可预期的问题。

类似于binlog由多个记录数据库修改的binlog文件和一个管理这些文件的index文件组成，relay-log由一系列包含了主库binlog事件的relay-log文件和一个管理这些文件的relay-log.index文件组成。relay-log文件具有和binlog文件一样的格式，同样可以使用mysqlbinlog工具来查看其中的内容。通过配置`relay-log="file-name"`和`relay-log-index="file-name"`这两个参数，可以指定relay-log文件和relay-log.index文件的名称。如果没有设置这两个参数，默认会使用主机名来作为relay-log和relay-log.index文件名的前缀。因此，当你使用主机名来命名relay-log时，在复制中途更改主机名可能导致因找不到relay-log文件而引发错误。

下面我们给出了relay-log会发生切换的几种情况。

- slave IO线程启动的时候。这发生在执行`START SLAVE`语句或者MariaDB/MySQL启动时。
- 执行`FLUSH LOGS`语句刷新日志时。
- 达到参数`max_relay_log_size`指定的大小时。当`max_relay_log_size`为0时，以参数`max_binlog_size`的值作为`max_relay_log_size`的值。

slave SQL线程在重放完一个relay-log文件中的所有事件时，会自动删除该relay-log文件，所以没有显示删除relay-log的命令。

8.3.3　master.info文件和relay-log.info文件

开启复制功能时，在从库的数据目录下会创建一个master.info文件和一个relay-log.info文件，它们用来记录复制的工作进度。

- master.info文件：该文件用来保存主库的主机名和端口信息以及登录到主库所需要的账号和密码。这里需要注意的是，账号和密码都是以文本的格式保存在master.info文件中，所以在实际应用中需要特别注意这一点，以防出现安全问题。在8.4节中我们会提到，最安全的做法是为复制单独创建一个账号，仅仅赋予该账号`REPLICATION SLAVE`权限。master.info文件还以binlog文件名和偏移量的形式记录了从库接收主库binlog事件的进度信息，有了这些信息，slave IO线程就知道下次该从哪里开始获取主库的binlog事件。
- relay-log.info文件：该文件用来记录从库的重放进度，所以在停止复制然后重新启动复制的时候，slave SQL线程知道该从哪里重新开始自己的工作。

配置`master-info-file="file-name"`和`relay-log-info-file="file-name"`参数，可以改变master.info文件和relay-log.info文件的名称。

在MySQL 5.6以及MariaDB 10.0中，还可以通过配置`master-info-repository=TABLE`（默认为`FILE`）来使用表`mysql.slave_master_info`代替master.info文件，存储复制相关的信息。同样，通过配置`relay-log-info-repository=TABLE`，可以用表`mysql.slave_relay_log_info`来代替relay-log.info文件。

正是有了master.info文件和relay-log.info文件，我们才能方便地使用`STOP SLAVE`停止复制工作，在进行一些备份或者计算等工作之后使用`START SLAVE`重新恢复复制工作。

8.4 复制的配置

在这一节中，我们将介绍在实际应用中应该如何配置MariaDB/MySQL的复制功能。

8.4.1 在新安装的主库和从库上配置复制

通常，配置MariaDB/MySQL的复制功能很简单，但由于应用场景的不同而存在一些差异。如果是对新安装的主库和从库配置复制功能，基本可以分为以下4步。

(1) 配置主库。
(2) 配置从库。
(3) 创建用于复制的账号。
(4) 开启复制。

下面详细介绍这4个步骤。

1. 配置主库

在主库上，首先必须开启你的binlog，其次必须配置一个唯一的`server_id`。如果没有配置这些参数，需要配置它们之后重启你的MariaDB/MySQL，保证配置生效。

前面我们提到，MariaDB/MySQL的复制功能是基于binlog的，从库接收主库的binlog事件，然后在本地进行重放，达到和主库拥有相同数据的目的，如果主库没有开启binlog，那么复制功能将不可用。

在一组复制结构中，每个服务器都必须配置一个唯一的server_id，它标识了一组复制结构中的一台MariaDB/MySQL服务器。该值必须是1到$2^{32}-1$之间的一个数。

如果你没有给主库配置server_id，将会使用默认值0，此时MariaDB/MySQL将拒绝所有从库的连接请求，这将导致复制功能不可用。

为了保证数据的安全性，你应该设置sync_binlog参数的值为1。如果你使用的是InnoDB，同时建议将参数innodb_flush_log_at_trx_commit设置为1。前者保证每往binlog中写入一个事务后立即将binlog持久化到磁盘，后者保证每往InnoDB存储引擎提交一个事务，立即将重做日志持久化到磁盘。如果你没有配置sync_binlog=1，写入到binlog文件的部分内容可能还存在于操作系统的页面缓存中，没有来得及持久化到磁盘，如果这个时候发生了宕机，将会丢失binlog的部分数据。

配置主服务器my.cnf中的log_bin和server_id的代码如下：

```
[mysqld]
log_bin = mysql-bin
server_id = 1
```

2. 配置从库

在从服务器上，你必须配置和主库不同的server_id，如果你没有进行相关的配置，或者当前的server_id和主库的server_id冲突，需要停止MariaDB/MySQL，更改配置，然后重新启动MariaDB/MySQL，使参数生效。

如果你设置了多个从服务器，那么每个从服务器必须配置不同的server_id。server_id区分了一组复制结构中的不同服务器。

如果你的从服务器没有配置server_id参数，默认值是0，这样从服务器会拒绝和主服务器建立连接。复制功能将不可用。

在从服务器上，你可以选择不开启binlog。但当你想把从服务器配置成其他服务器的主服务器时，必须开启binlog。当开启了从库的binlog时，如果想把重放的事件也记录到binlog中，可以将参数log_slave_updates配置成1。

配置从服务器my.cnf的server_id的代码如下：

```
[mysqld]
server_id = 2
```

3. 创建复制账号

在复制之前，必须建立一个从库到主库的连接才能进行相应的数据传输，所以必须在主服务器上为该连接创建一个账号，允许从库连接到主服务器上，并且赋予该账号复制权限。

请尽可能地使用只有REPLICATION SLAVE权限的账号进行复制，因为如果你使用的账户具有其他权限，则当你开启复制功能之后，从服务器会将你所使用的复制账号和密码以文本的格式记录到master.info文件中，这样会存在一定的安全隐患。

在主库上使用CREATE USER以及GRANT命令创建用于复制的账号并且赋予该账号相关的权限，相关代码如下：

```
mysql> CREATE USER "slave_usr"@"%" IDENTIFIED BY "passwd";
mysql> GRANT REPLICATION SLAVE ON *.* TO "slave_usr"@"%";
```

如果你想防止别人拿着该账号在其他地方复制，那么请限制该账号的使用范围。例如,你只想该账号在IP为192.168.1.12的机器上使用，那么创建复制账号时可以采用CREATE USER "slave_user"@"192.168.1.12" IDENTIFIED BY 'passwd'命令。

4. 开启复制功能

在配置完主库和从库,并且创建了具有复制权限的账号之后,接下来就可以开启复制功能了。

通过在从库上执行CHANGE MASTER TO命令来指定或者更改（如果之前已经调用过CHANGE MASTER TO命令）主库的信息。

CHANGE MASTER TO命令的基本语法如下所示：

```
CHANGE MASTER TO MASTER_HOST="host_name",
    MASTER_USER = "rpl-user",
    MASTER_PASSWORD ="rpl-passwd",
    MASTER_PORT = port,
    MASTER_DELAY = interval,
    MASTER_LOG_FILE = "master-log-file",
    MASTER_LOG_POS = master-log-pos;
```

其中MASTER_HOST、MASTER_PORT、MASTER_USER和MASTER_PASSWORD指定了连接主库需要的所有信息。MASTER_LOG_FILE和MASTER_LOG_POS指定了从库在什么位置开始接收主库的binlog事件。MASTER_DELAY指定了从库的数据应该比主库数据延时多久。

执行CHANGE MASTER TO命令并不会连接到主库，仅仅是将这些信息写入到从库数据目录下的master.info文件中。如果MASTER_LOG_FILE没有指定，则默认是空，因为这个时候没有和主库建立连接，还不知道主库的binlog文件名称，只有等到和主库建立连接时才能得知。如果MASTER_LOG_POS没有指定，则默认值是4，这样就略过了binlog的头4个字节（binlog文件头，4个

字节的常量），从第一个事件开始读取。

接下来就是通过调用START SLAVE命令来开启复制功能。该命令开启了slave IO和slave SQL两个线程，前者负责建立从库到主库的连接，接收主库的binlog事件，然后写入到relay-log中，后者负责读取relay-log的内容进行重放。

下面我们给出了相应的操作示例：

```
mysql> CHANGE MASTER TO MASTER_HOST="127.0.0.1", MASTER_PORT=3306, MASTER_USER="slave",
    MASTER_PASSWORD="slave";
Query OK, 0 rows affected, 2 warnings (0.33 sec)

mysql> START SLAVE;
Query OK, 0 rows affected (0.05 sec)
```

此时在主库上可以看到负责发送binlog事件的master dump线程正在运行，如下所示：

```
mysql> SHOW PROCESSLIST;
+-------+--------+-------------------+-------+---------------+------+---------------- ---+
| Id    | User   | Host              | ...   | Command       | Time | State              |
+-------+--------+-------------------+-------+---------------+------+---------------- ---+
| 1     | root   | localhost:56608   | ...   | Query         | 0    | init               |
| 4     | slave  | localhost:56642   | ...   | Binlog Dump   | 11   | Master has sent... |
+-------+--------+-------------------+-------+---------------+------+--------------  ----+
2 rows in set (0.00 sec)
```

在从库上可以看到slave IO线程和slave SQL线程也在运行，如下所示：

```
mysql> SHOW PROCESSLIST;
+-------+---------------+----------------+------+----------+------+-----------------+
| Id    | User          | Host           | ...  | Command  | Time | State           |
+-------+---------------+----------------+------+----------+------+-----------------+
| 1     | root          | localhost:60029| ...  | Query    | 0    | init            |
| 6     | system user   |                | ...  | Connect  | 1    | Waiting for...  |
| 7     | system user   |                | ...  | Connec   | 0    | Slave has read... |
+-------+---------------+----------------+------+----------+------+-----------------+
3 rows in set (0.00 sec)
```

8.4.2　主库有一定数据时的复制配置

上面我们介绍的是在新安装的主库和从库上配置复制功能，由于主库里面没有任何数据，配置相对简单。如果当主库有一定的数据时，情况会稍微复杂一点，基本的步骤如下。

(1) 配置主库和从库。
(2) 创建用于复制的账号。
(3) 创建主库的快照，获取主库状态。

(4) 开启复制功能。

前两步我们已经介绍完了，这里主要介绍后面两步。

1. 创建主库的快照

使用mysqldump工具能够方便地创建主库的快照，基本步骤如下。

(1) 在主库上执行FLUSH TABLES WITH READ LOCK，让主库处于只读状态：

```
mysql> FLUSH TABLES WITH READ LOCK;
```

(2) 在主库上执行SHOW MASTER STATUS命令获取当前binlog的信息，包括当前binlog文件的名称以及当前写到binlog文件的哪个位置。这些信息将作为从库执行CHANGE MASTER TO命令时参数MASTER_LOG_FILE和MASTER_LOG_POS的值：

```
mysql> SHOW MASTER STATUS;
+------------------------------------+-----------------+---------+
| File                               | Position        | ...     |
+------------------------------------+-----------------+---------+
| mysql-bin.000005                   | 1687            | ...     |
+------------------------------------+-----------------+---------+
1 row in set (0.00 sec)
```

(3) 在shell命令行中执行mysqldump命令将指定的数据库或者表的数据导出来（如果你只想复制主库的某些库或者某些表的数据）：

```
shell> mysqldump [options] database_name [table_naname] > dump.file
```

(4) 在主库上执行UNLOCK TABLES命令，解除主库的只读锁：

```
mysql > UNLOCK TABLES;
```

(5) 使用dump.file文件在从库上建立快照：

```
shell > mysql -u root -p -h192.168.1.12 -P3306 < dump.file
```

当我们使用mysqldump命令时，如果添加了--master-data参数，是不需要对主库执行FLUSH TABLES WITH READ LOCK命令来使主库处于只读状态的，也不需要执行SHOW MASTER STATUS来获取主库当前binlog的相关信息。因为如果携带了--master-data参数，mysqldump内部会自动执行FLUSH TABLES WITH READ LOCK来使库处于只读状态，然后执行SHOW MASTER STATUS来获取当前binlog的信息，并将其记录到输出文件中。当mysqldump完成快照之后，会自动释放获取的全局读锁。

mysqldump命令主要用于导出数据库的库表结构和数据，用于对数据库进行备份或者复制。下面列出了mysqldump --help的部分输出结果：

```
jinpengzhang@jinpengzhang:~$ mysqldump --help
Dumping structure and contents of MySQL databases and tables.
Usage: mysqldump [OPTIONS] database [tables]
OR     mysqldump [OPTIONS] --databases [OPTIONS] DB1 [DB2 DB3...]
OR     mysqldump [OPTIONS] --all-databases [OPTIONS]

Default options are read from the following files in the given order:
/etc/my.cnf /etc/mysql/my.cnf /usr/etc/my.cnf ~/.my.cnf

The following options may be given as the first argument:
--print-defaults      Print the program argument list and exit.
--no-defaults         Don't read default options from any option file.
--defaults-file=#     Only read default options from the given file #.
--defaults-extra-file=# Read this file after the global files are read.
  -A, --all-databases Dump all the databases. This will be same as --databases
                      with all databases selected.
  -Y, --all-tablespaces
                      Dump all the tablespaces.
  -y, --no-tablespaces
                      Do not dump any tablespace information.
  --add-drop-database Add a DROP DATABASE before each create.
  --add-drop-table    Add a DROP TABLE before each create.
                      (Defaults to on; use --skip-add-drop-table to disable.)
  --add-locks         Add locks around INSERT statements.
                      (Defaults to on; use --skip-add-locks to disable.)
  --allow-keywords    Allow creation of column names that are keywords.
  --apply-slave-statements
                      Adds 'STOP SLAVE' prior to 'CHANGE MASTER' and 'START
                      SLAVE' to bottom of dump.
  --character-sets-dir=name
                      Directory for character set files.
  -i, --comments      Write additional information.
                      (Defaults to on; use --skip-comments to disable.)
  -c, --complete-insert
                      Use complete insert statements.
  -C, --compress      Use compression in server/client protocol.
  -a, --create-options
                      Include all MySQL specific create options.
                      (Defaults to on; use --skip-create-options to disable.)
  -B, --databases     Dump several databases. Note the difference in usage; in
                      this case no tables are given. All name arguments are
                      regarded as database names. 'USE db_name;' will be
                      included in the output.
  --default-character-set=name
                      Set the default character set.
  --delete-master-logs
                      Delete logs on master after backup. This automatically
                      enables --master-data.
  --dump-slave[=#]    This causes the binary log position and filename of the
```

```
                    master to be appended to the dumped data output. Setting
                    the value to 1, will print it as a CHANGE MASTER command
                    in the dumped data output; if equal to 2, that command
                    will be prefixed with a comment symbol. This option will
                    turn --lock-all-tables on, unless --single-transaction is
                    specified too (in which case a global read lock is only
                    taken a short time at the beginning of the dump - don't
                    forget to read about --single-transaction below). In all
                    cases any action on logs will happen at the exact moment
                    of the dump.Option automatically turns --lock-tables off.
-e, --extended-insert
                    Use multiple-row INSERT syntax that include several
                    VALUES lists.
                    (Defaults to on; use --skip-extended-insert to disable.)
-F, --flush-logs    Flush logs file in server before starting dump. Note that
                    if you dump many databases at once (using the option
                    --databases= or --all-databases), the logs will be
                    flushed for each database dumped. The exception is when
                    using --lock-all-tables or --master-data: in this case
                    the logs will be flushed only once, corresponding to the
                    moment all tables are locked. So if you want your dump
                    and the log flush to happen at the same exact moment you
                    should use --lock-all-tables or --master-data with
                    --flush-logs.
-?, --help          Display this help message and exit.
--hex-blob          Dump binary strings (BINARY, VARBINARY, BLOB) in
                    hexadecimal format.
-h, --host=name     Connect to host.
--ignore-table=name Do not dump the specified table. To specify more than one
                    table to ignore, use the directive multiple times, once
                    for each table.  Each table must be specified with both
                    database and table names, e.g.,
                    --ignore-table=database.table.
--include-master-host-port
                    Adds 'MASTER_HOST=<host>, MASTER_PORT=<port>' to 'CHANGE
                    MASTER TO..' in dump produced with --dump-slave.
-x, --lock-all-tables
                    Locks all tables across all databases. This is achieved
                    by taking a global read lock for the duration of the
                    whole dump. Automatically turns --single-transaction and
                    --lock-tables off.
-l, --lock-tables   Lock all tables for read.
                    (Defaults to on; use --skip-lock-tables to disable.)
--master-data[=#]   This causes the binary log position and filename to be
                    appended to the output. If equal to 1, will print it as a
                    CHANGE MASTER command; if equal to 2, that command will
                    be prefixed with a comment symbol. This option will turn
                    --lock-all-tables on, unless --single-transaction is
                    specified too (in which case a global read lock is only
```

```
                    taken a short time at the beginning of the dump; don't
                    forget to read about --single-transaction below). In all
                    cases, any action on logs will happen at the exact moment
                    of the dump. Option automatically turns --lock-tables
                    off.
--no-autocommit     Wrap tables with autocommit/commit statements.
-n, --no-create-db  Suppress the CREATE DATABASE ... IF EXISTS statement that
                    normally is output for each dumped database if
                    --all-databases or --databases is given.
-d, --no-data       No row information.
-N, --no-set-names  Same as --skip-set-charset.
--order-by-primary  Sorts each table's rows by primary key, or first unique
                    key, if such a key exists.  Useful when dumping a MyISAM
                    table to be loaded into an InnoDB table, but will make
                    the dump itself take considerably longer.
-p, --password[=name]
                    Password to use when connecting to server. If password is
                    not given it's solicited on the tty.
-P, --port=#        Port number to use for connection.
--protocol=name     The protocol to use for connection (tcp, socket, pipe,
                    memory).
-q, --quick         Don't buffer query, dump directly to stdout.
                    (Defaults to on; use --skip-quick to disable.)
-Q, --quote-names   Quote table and column names with backticks (`).
                    (Defaults to on; use --skip-quote-names to disable.)
-r, --result-file=name
                    Direct output to a given file. This option should be used
                    in systems (e.g., DOS, Windows) that use carriage-return
                    linefeed pairs (\r\n) to separate text lines. This option
                    ensures that only a single newline is used.
--single-transaction
                    Creates a consistent snapshot by dumping all tables in a
                    single transaction. Works ONLY for tables stored in
                    storage engines which support multiversioning (currently
                    only InnoDB does); the dump is NOT guaranteed to be
                    consistent for other storage engines. While a
                    --single-transaction dump is in process, to ensure a
                    valid dump file (correct table contents and binary log
                    position), no other connection should use the following
                    statements: ALTER TABLE, DROP TABLE, RENAME TABLE,
                    TRUNCATE TABLE, as consistent snapshot is not isolated
                    from them. Option automatically turns off --lock-tables.
--ssl               Enable SSL for connection (automatically enabled with
                    other flags).
-T, --tab=name      Create tab-separated textfile for each table to given
                    path. (Create .sql and .txt files.) NOTE: This only works
                    if mysqldump is run on the same machine as the mysqld
                    server.
--tables            Overrides option --databases (-B).
```

```
  -u, --user=name     User for login if not current user.
  -v, --verbose       Print info about the various stages.
  -V, --version       Output version information and exit.
  -X, --xml           Dump a database as well formed XML.
...
```

mysqldump通常的使用方式有如下3种：

```
mysqldump [options] database [tables]
mysqldump [options] --database [options] db1 [db2 db3 ...]
mysqldump [options] -all-databases [options]
```

第一种方式主要用于导出单个数据库或者导出单个数据库的某些表；第二种方式主要用于同时导出多个数据库；而第三种方式是导出整个数据库的内容。

如果你不为mysqldump指定参数，它会自动从相关的配置文件中读取配置。自动读取配置文件的顺序为：/etc/my.cnf，/etc/mysql/my.cnf，/usr/etc/my.cnf ~/.my.cnf。

--master-data参数用于输出binlog的文件名和当前位置。如果该参数的值为1，会将binlog文件名以及当前binlog的位置信息作为CHANGE MASTER TO语句的参数输出。如果将该参数设置为2，CHANGE MASTER TO语句会被写成SQL注释。--master-data参数会获取数据库的全局读锁，让数据库处于只读状态，在mysqldump执行结束后会自动释放该全局读锁，恢复数据库的写。当同时携带了--single-transaction参数的时候，--master-data不会获取全局读锁。

下面我们给出使用--master-data参数的例子：

```
/*--------------------------------------example8-1--------------------------------------------*/

shell> mysqldump -h192.168.1.11 -uroot -P3306 --master-data=1 db1 >dump.file1
shell> less dump.file1
...
--
-- Position to start replication or point-in-time recovery from
--

CHANGE MASTER TO MASTER_LOG_FILE="mysql-bin.000005", MASTER_LOG_POS=1687;
...

/*--------------------------------------example8-2--------------------------------------------*/

shell> mysqldump -h192.168.1.11 -uroot -P3306 --master-data=2 db2 >dump.file2
shell> less dump.file2
...
--
-- Position to start replication or point-in-time recovery from
--
```

```
-- CHANGE MASTER TO MASTER_LOG_FILE="mysql-bin.000005", MASTER_LOG_POS=1687;
...
```

--single-transaction参数仅仅适用于支持MVCC机制的事务存储引擎的表，例如InnoDB的表。当携带了--single-transaction参数时，mysqldump的执行流程如下。

(1) 执行FLUSH TABLES WITH READ LOCK语句获取全局读锁，让数据库处于只读状态。

(2) 设置当前会话的事务隔离级别为RR，执行START TRANSACTION语句开启一个新事务。

(3) 如果同时携带了--master-data参数，执行SHOW MASTER STATUS语句获取当前binlog的名称以及位置信息。

(4) 执行UNLOCK TABLES语句释放获取的全局读锁，这样其他事务就可以对数据库进行写操作。

(5) 在开启的事务内部进行数据库的导出操作。

(6) 结束事务。

携带了--single-transaction参数的mysqldump命令仅仅会占用全局读锁很短的时间，几乎不会阻塞其他事务对数据库的写操作。同时由于它是在事务内部进行数据库的导出操作的，这保证了能够获取数据库一个完整的镜像。

2. 开启复制功能

现在从库已经有了数据，并且也获得了主库的binlog信息，这时我们需要做的就是执行CHANGE MASTER TO和START SLAVE命令来开启复制功能，其中 MASTER_LOG_FILE和 MASTER_LOG_POS这两个参数就是上一步中获取的主库当前binlog文件名以及位置信息，而复制将从这两个参数指定的位置开始进行。

8.4.3　选择性复制

MariaDB/MySQL提供了库和表两种粒度的复制过滤功能。设置相应的参数，可以让MariaDB/MySQL只复制指定的一些库或表，或者忽略指定的一些库或表。

在主库上配置binlog-do-db参数和binlog-ignore-db参数来控制哪些库的更新需要记录在binlog中，这直接影响到从库将拥有的数据内容。没有记录在binlog中的更改是无法复制到从库上的，通常情况下尽量不要开启该选项，特别是启用复制功能的时候，不然可能会出现从库上有些数据莫名没有的情况。

通过设置replicate-do-db参数和replicate-ignore-db参数，可以指定需要对哪些库进行复制，或者对哪些库不进行复制。如果想指定复制多个库或者忽略多个库，可以配置多个replicate-do-db参数和replicate-ignore-db参数。

replicate-do-table参数和replicate-ignore-table参数提供了表级别的选择性复制，这两个

参数同样可以设置多个，表示对多个表进行选择性复制或选择性忽略。

参数replicate-rewrite-db=*from_name*->*to_name*告诉从库在重放过程中，如果该事件在主库上执行时默认的数据库是*from_name*，将从库的当前默认库改成*to_name*。这个转换动作发生在匹配过滤条件之前。replicate-rewrite-db参数只会对操作表级别的语句生效，不会对create database、drop database、alter database等操作数据库的语句起作用。该参数也可以设置多个，表示多个转换关系。

replicate-wild-do-table参数和replicate-wild-ignore-table参数允许你对特定模式的表进行选择性复制或者选择性忽略。例如：设置replicate-wild-do-table=bj%.foo%，那么bj1.foo1、bj.foo、bjx.fooy等之类的表都会匹配上。参数replicate-wild-do-table和参数replicate-wild-ignore-table同样可以设置多个。

图8-2很好地描述了复制过滤器的工作原理。

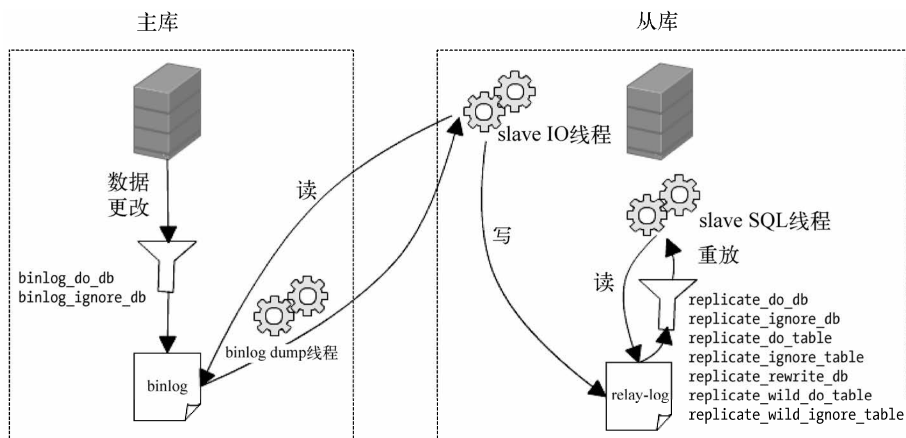


图8-2　复制过滤器

通常情况下，如果从库开启了binlog，从库重放的内容是不会记录到binlog中去的。如果从库本身存在其他从库，比如A->B->C这种情况，B是A的从库，同时B又是C的主库，那么可以通过将B的log-slave-updates参数置为1将B库slave SQL线程重放的内容也记录到B的binlog中去，这样C就能体现A的内容更新情况。

8.5　复制的实现

在本节中，我们将从源代码实现的角度来讲解MariaDB/MySQL的复制功能，其中列出了一些重要的数据结构和函数的实现。为了方便阅读，部分函数的参数可能会省略，但这不影响对整个复制机制的理解。

作为MySQL的一个分支，MariaDB在很多方面都保留了MySQL原有的实现。MariaDB的复制功能也沿用了MySQL的实现，在此基础上进行了一些扩展，例如支持多源复制。所以我们首先从MySQL的复制实现来进行分析，当遇到MariaDB的不同之处时会特别说明。

8.5.1　复制相关的数据结构

首先我们给出与复制相关的几个核心数据结构的定义，并且详细介绍其中每个成员的具体含义。

类Rpl_info定义了复制相关的一些通用信息，例如slave线程的上下文信息以及运行状况，等等。类Master_info和类Relay_log_info都继承自Rpl_info类，前者主要记录连接到主库所需要的信息以及从库接收binlog的进度等信息，后者用于保存从库重放binlog的进度以及其他一些相关的信息。

1. Rpl_info

类Rpl_info定义了复制相关的一些通用信息。

类Rpl_info存在于MySQL 5.6中，而在MariaDB 10.0中看不到这个类，因为该类的成员被移到了Master_info和Relay_log_info中，但成员的含义保持不变。

下面给出了MySQL 5.6中类Rpl_info的定义：

```
// sql/rpl_info.h

class Rpl_info
{
public:
    THD *info_thd;                 // 线程的上下文环境
    bool inited;                   // 初始化标志
    volatile bool abort_slave;     // 是否停止slave线程
    volatile uint slave_running;   // slave线程是否在运行

    ...
protected:
    Rpl_info_handler *handler;
};
```

下面我们给出Rpl_info各个成员的具体含义。

- info_thd：在前面的章节中已经介绍过，在MariaDB/MySQL中，每个线程都会对应一个THD类型的实例，用于存储线程相关的上下文信息。info_thd变量代表了slave IO/slave SQL线程的执行上下文。
- inited：是否进行了初始化。

- abort_slave：如果为1，表示需要停止slave线程。
- slave_running：该成员用来标识slave线程是否在运行。例如当Master_info中该成员的值为1时，表示slave IO线程正在运行，当Relay_log_info中该成员的值为1时，表示slave SQL线程正在运行。在C/C++中，关键字volatile提醒编译器它修饰的变量随时可能改变，因此编译后的程序每次需要读取或者存储该变量时，都会去内存中读取它的值，而不是从寄存器中读取，以防止一个线程对该变量进行写操作而另一个线程对该线程进行读取操作时，两个线程的数据不一致。
- handler：该成员主要用于对Master_info实例的内容和master.info文件或mysql.slave_master_info表的内容进行同步以及对Relay_log_info实例的内容和relay_log.info文件或mysql.slave_relay_log_info表的内容进行同步。当mysqld启动时，从文件或表中读取内容到Master_info/Relay_log_info实例中，当Master_info/Relay_log_info实例的内容有更新时，写入到对应的文件或表中进行持久化。

2. Master_info

下面我们给出类Master_info的定义。该类主要记录了连接到主库所需要的信息以及接收binlog的进度等信息，它在MySQL中只有一个实例，那就是全局变量active_mi，该变量在mysqld启动阶段由init_slave函数进行初始化。由于MariaDB支持多源复制，允许一台从库拥有多个主库，所以在MariaDB上可能存在多个Master_info类的实例，通过Master_info_index类来管理，这在8.8节中会进一步分析。

下面给出类Master_info的定义：

```
// sql/rpl_mi.h

class Master_info : public Rpl_info
{
public:
    char host[HOSTNAME_LENGTH + 1];           // 主库主机名或IP地址
    uint port;                                // 主库的端口号
    char bind addr[HOSTNAME_LENGTH+1];
private:
    bool start_user_configured;               // 如果使用start slave指定用户名和密码，则该变量为true
    char user[USERNAME_LENGTH + 1];           // master.info文件中的用户名
    char password[MAX_PASSWORD_LENGTH + 1];   // master.info文件中的密码
    char start_user[USERNAME_LENGTH + 1];     // 执行start slave命令时指定的用户名
    char start_password[MAX_PASSWORD_LENGTH + 1];   // 执行start slave命令时指定的密码
    char start_plugin_auth[FN_REFLEN + 1];    // 传输密码使用的插件
    char start_plugin_dir[FN_REFLEN + 1];

    uint connect_retry;                       // 重连的最多次数

    MYSQL *mysql;                             // 和主库之间的连接
```

8

```
    long clock_diff_with_master;              // 与主库之间的时间差异
    float heartbeat_period;                   // 与主库心跳的间隔时间
    time_t last_heartbeat;                    // 最近一次心跳时间
    Relay_log_info *rli;                      // 指向Relay_log_info结构

    ulong master_id;                          // 主库的server_id
    char master_uuid[UUID_LENGTH+1];          // 主库的uuid
    ulong master_gtid_mode;                   // 主库是否支持GTID

    Format_description_log_event *mi_description_event;

    bool auto_position;                       // 如果为true并且支持GTID复制，则在执行CHANGE MASTER
                                              // TO命令时不需要指定binlog的位置

protected:
    // 以下两个变量指定从库的复制进度
    char master_log_name[FN_REFLEN];
    my_off_t master_log_pos;

   ...
};
```

下面我们给出Master_info各个成员的具体含义。

- host：主库的地址，通过CHANGE MASTER TO语句指定。
- port：主库的端口号，通过CHANGE MASTER TO语句指定。
- bind_addr：bind_addr的作用是当从库所在的机器有多个网络接口时，选择指定的接口与主库进行通信。
- user：连接到主库进行复制的账号，通过CHANGE MASTER TO语句指定。
- password：连接到主库所需的密码，通过CHANGE MASTER TO语句指定。
- start_user：执行START SLAVE命令时指定的账号。
- start_password：执行START SLAVE命令时指定的密码。
- master_id：主库的server_id，在slave IO线程和主库建立连接之后向主库询问获得。
- master_uuid：主库的uuid，在slave IO线程和主库建立连接之后向主库询问获得。
- mysql：从库调用START SLAVE命令开启复制时，会建立一个到主库的连接，mysql成员就是该连接。之后所有与主库的通信都是通过该连接进行的。
- clock_diff_with_master：在slave IO线程启动之后会获取主库的系统时间和时区等信息，clock_diff_with_master变量存储的是从库与主库之间的时间差。在执行CHANGE MASTER TO命令时如果设置了sql_delay参数，从库在对事件进行重放时，就能根据事件在主库上的执行时间以及clock_diff_with_master来决定该事件应该何时进行重放。
- heartbeat_period：heartbeat_period表示发送心跳包的间隔。当主库很长时间没有数据更新时，主库和从库之间通过发送心跳包来保持两者之间的连接。

- ❑ master_gtid_mode：表示在建立从库到主库的连接之后，询问主库是否开启了GTID模式。
- ❑ auto_position：如果该成员为true并且支持GTID复制，则执行CHANGE MASTER TO 命令时不需要指定binlog的位置。GTID相关的内容将会在8.9节中详细介绍。
- ❑ last_heartbeat：last_heart_beat表示最近一次收到心跳包的时间。
- ❑ master_log_name[FN_REFLEN]和my_off_tmaster_log_pos：master_log_name和master_log_pos成员记录了slave IO线程接收binlog事件的进度。

3. Relay_log_info

Relay_log_info主要用于记录slave SQL线程重放binlog的进度信息以及其他相关的信息。下面给出了Relay_log_info类的定义：

```
// sql/rpl_rli.h

class Relay_log_info : public Rpl_info
{
public:
    bool replicate_same_server_id;          // replicate-same-server-id选项

    MYSQL_BIN_LOG relay_log;                // relay-log
    File cur_log_fd;                        // 当前读取的relay-log文件的描述符
    IO_CACHE cache_buf,*cur_log;            // 读写文件的缓存

    bool is_relay_log_recovery;             // MySQL启动时是否进行relay-log修复工作
    Master_info *mi;                        // 指向Master_info对象

protected:
    // 一个事务包含多个binlog事件，这些事件称为一个组
    char group_relay_log_name[FN_REFLEN];
    ulonglong group_relay_log_pos;
    char event_relay_log_name[FN_REFLEN];
    ulonglong event_relay_log_pos;
    ulonglong future_event_relay_log_pos;

    char group_master_log_name[FN_REFLEN];
    volatile my_off_t group_master_log_pos;

    ulonglong future_group_master_log_pos;

private:
    // relay-log中包含的GTID集合
    Gtid_set gtid_set;

public:
    // relay-log所占磁盘空间相关信息
    ulonglong log_space_limit,log_space_total;
```

```
    bool ignore_log_space_limit;

    // 用于relay-log清理时slave SQL线程和slave IO线程通信
    mysql_mutex_t log_space_lock;
    mysql_cond_t log_space_cond;

    // slave SQL线程要求slave IO线程进行relay-log切换
    bool sql_force_rotate_relay;

    // 当前事务重试了多少次
    ulong trans_retries;
    // 从slave SQL线程开启到现在重试了多少次
    ulong retried_trans;


private:
    // 事件的延迟重放，对应于执行CHANGE MASTER TO命令时的master_delay参数
    int sql_delay;
    time_t sql_delay_end;

   ...
};
```

下面我们给出Relay_log_info各个成员的含义。

- replicate_same_server_id：该成员保存的是replicate-same-server-id选项的值，如果该选项的值为1，从库将不会过滤与自己拥有相同server_id的事件。
- cur_log_fd：当前打开的relay-log文件的描述符。
- relay_log：用于操作relay-log。relay-log的格式和结构与binlog完全一致，所以使用了类MYSQL_BIN_LOG对relay-log进行操作。
- cache_buf和cur_log：在之前的章节中我们已经介绍过，在MariaDB/MySQL中，所有的文件读写都是通过IO_CACHE进行缓存的。成员cache_buf和cur_log是relay-log文件的缓存。在relay-log中，正在被slave IO线程写的relay-log文件称为活跃的relay-log文件，其他的relay-log文件叫非活跃的relay-log文件。当重放的是非活跃的relay-log文件时，cache_buf对应于打开的非活跃的relay-log文件，cur_log指向cache_buf；当重放的是活跃的relay-log文件时，cache_buf为空，cur_log对应于slave IO线程打开的活跃的relay-log文件。
- is_relay_log_recovery：该成员对应于relay-log-recovery选项。如果指定了该选项，MariaDB/MySQL在进行初始化时会对relay-log进行相应的裁剪，将之前获取了但是没有重放的事件从relay-log中丢弃掉，这部分事件将从主库重新获取。当从库发生宕机，导致relay-log损坏，一部分的relay-log没有处理时，如果使用relay-log-recovery选项，会自动放弃所有未执行的relay-log，并且重新从主库上获取binlog事件，这样能够保证relay-log的完整性。

- group_relay_log_name：一个事务通常包含多个binlog事件，这些事件称为一个组。group_relay_log_name表示当前重放的事务所在的relay-log文件的名称。
- group_relay_log_pos：当前重放的事务在relay-log文件中的起始位置。
- event_relay_log_name：当前重放的事件所在的relay-log文件的名称。
- event_relay_log_pos：当前重放的事件在relay-log文件中的位置。
- future_event_relay_log_pos：下一个事件在relay-log文件中的位置。
- group_master_log_name：当前重放的事务在主库上对应的binlog文件的名称。
- group_master_log_pos：当前重放的事务在主库上binlog文件中的位置。
- gtid_set：gtid_set是从库relay-log中所包含的GTID集合。GTID相关的内容在8.9节中详细介绍。
- sql_force_rotate_relay：当slave SQL线程想删除一些relay-log文件来腾出磁盘空间时，可能会要求slave IO线程执行relay-log的切换。如果sql_force_rotate_relay为1，表明slave SQL线程要求slave IO线程进行relay-log的切换。
- log_space_limit：relay-log占用磁盘空间大小的上限。
- log_space_total：relay-log占用的磁盘空间的大小。
- sql_delay：指用户在调用CHANGE MASTER TO命令时指定的从库的延迟执行时间，单位为秒。
- sql_delay_end：指当前被延迟的事件应该何时重放。
- mi：mi指向Master_info的实例。

除了上面列出的一些成员外，Relay_log_info还包含了许多其他成员，例如用于实现并发复制的相关成员、用于实现条件停止复制（until condition replication）的一些成员，等等。

8.5.2 复制的初始化——init_slave函数

在mysqld启动的时候，会调用init_slave函数对复制功能进行初始化。下面列出了init_slave函数的主要代码：

```
1. // sql/rpl_slave.cc

2. #define GOTO_ERR { error = 1; goto err; }
3. int init_slave() {
4.     int error = 0;
5.     int thread_mask = SLAVE_SQL | SLAVE_IO;
6.     enum_return_check check_return = ERROR_CHECKING_REPOSITORY;
7.     Relay_log_info *rli = NULL;

       // 创建Master_info对象
8.     active_mi = Rpl_info_factory::create_mi(opt_mi_repository_id);
9.     if (!active_mi)  GOTO_ERR;

       // 创建Relay_log_info对象
10.    rli = Rpl_info_factory::create_rli(opt_rli_repository_id);
```

```
11.     if (!rli)  GOTO_ERR;

        // 将Master_info对象和Relay_log_info对象进行关联
12.     active_mi->set_relay_log_info(rli);
13.     rli->set_master_info(active_mi);

        // 初始化Master_info对象
14.     check_return = active_mi->check_info();
15.     if (check_return != REPOSITORY_DOES_NOT_EXIST && (thread_mask & SLAVE_IO) && active_mi->mi_
            init_info())
16.         GOTO_ERR;

        // 初始化Relay_log_info对象
17.     check_return = active_mi->rli->check_info();
18.     if (check_return == REPOSITORY_DOES_NOT_EXIST && (thread_mask & SLAVE_SQL) && active_mi-
            >rli->rli_init_info())
19.         GOTO_ERR;

        // 启动slave线程
20.     if (active_mi->host[0] && !opt_skip_slave_start) {
21.         if (start_slave_threads(true/*need_lock_slave=true*/,
22.                 false/*wait_for_start=false*/, active_mi, thread_mask))
23.                 GOTO_ERR;
24.     }

25.err:
26.     if (err) sql_print_information("init slave failed!");
27.     return err;
28.}
```

接下来，我们针对上面给出的源代码来分析init_slave函数所做的工作。

第8行代码调用Rpl_info_factory::create_mi函数创建全局变量Master_info *active_mi。根据配置的master-info-repository是FILE还是TABLE，将active_mi的handler成员初始化为合适的实例。此外，create_mi函数还做了一件事情：如果master info存在（指的是master.info文件存在或者表mysql.slave_master_info中存在数据），也就是说之前配置过复制，并且本次master-info-repository配置的值和上次不一样，比如上一次配置的是TABLE，这一次配置的是FILE，该函数会将mysql.slave_master_info表中的内容复制到master.info文件中，然后将mysql.slave_master_info表中的内容清空。

第10行代码调用Rpl_info_factory::create_rli函数创建一个Relay_log_info实例。

第12行和第13行代码将类Master_info的实例active_mi和类Relay_log_info的实例rli关联起来。此后访问复制相关的信息，只需要通过全局变量active_mi就可以了。

第14行和第15行代码说明，如果master.info文件或mysql.master_info表中有内容，调用函数

active_mi->mi_init_info初始化Master_info的成员，这些成员包括host、port、master_log_name、master_log_pos等。

第17行和第18行代码说明，如果relay-log.info文件或mysql.relay_log_info表中有内容，调用函数active_mi->rli->rli_init_info初始化Relay_log_info的相关成员。然后新建一个relay-log文件来记录将要从主库接收的binlog事件，也就是说每次启动mysqld时relay-log会发生一次切换。

第20行和第21行代码说明，如果之前配置过复制，并且启动的时候没有指定skip_slave_start选项，则启动slave IO线程和slave SQL线程开始复制工作。

至此，init_slave函数的主要工作就完成了，此时的复制工作就交给了slave IO和slave SQL两个线程，后续我们会对它们进行详细的分析。

在后面实现细节的介绍过程中，我们假设master-info-repository和slave-relay-info-repository参数设置的都是FILE，也就是说采用master.info文件和relay_log.info文件来存储复制的相关信息。如果参数master-info-repository和slave-relay-info-repository设置为TABLE，则处理流程基本一致。

8.5.3 CHANGE MASTER TO命令——准备工作

在开启从库的复制之前，需要调用CHANGE MASTER TO命令来指定主库的位置以及连接到主库的账号、密码等信息。本节中，我们将详细讲解CHANGE MASTER TO命令是如何工作的，具体做了哪些事情。下面给出了CHANGE MASTER TO命令的执行流程。

(1) 检查slave IO线程和slave SQL线程是否在运行，如果任意一个在运行，输出错误信息并结束命令的执行，否则继续往下执行。

(2) 检查MASTER-HOST参数输入的是否为空，如果为空，输出错误信息并结束执行，否则继续往下执行。

(3) 检查master.info文件是否存在。

  (a) 如果不存在就创建该文件，将Master_info的实例active_mi初始化为默认值，并将active_mi的信息同步到master.info文件中。

  (b) 如果master.info文件存在，读取master.info文件的信息到Master_info实例active_mi中。

(4) 检查relay_log.info文件是否存在。

  (a) 如果不存在，创建relay_log.info文件，同时创建relay-log相关的文件relay- log.index和relay-log.000001。将Relay_log_info的实例active_mi->rli初始化为默认值，同时将实例active_mi->rli的信息同步到relay_log.info文件中。

  (b) 如果relay_log.info文件存在，从中读取信息到实例active_mi->rli。

(5) 如果参数MASTER_HOST或MASTER_PORT和原来的不一样，也就是说如果主库变成另一个

MariaDB/MySQL实例，重置active_mi的master_uuid成员和master_id成员。将参数中的MASTER_LOG_NAME、MASTER_LOG_POS、MASTER_USER以及MASTER_PASSWORD等设置到实例active_mi中。

(6) 如果指定了参数RELAY_LOG_FILE和RELAY_LOG_POS等信息，则定位到RELAY_LOG_FILE和RELAY_LOG_POS指定的位置。

(7) 将active_mi的内容同步到master.info文件中，将active_mi->rli的内容同步到relay_log.info文件中。

通过上面的介绍，我们发现CHANGE MASTER TO命令仅仅是做了一些准备工作，此时复制工作还没有正式开始，slave IO线程以及slave SQL线程也没有开启。要开启复制工作，必须执行START SLAVE命令。

8.5.4 START SLAVE命令——开启复制

该命令主要用于开启slave IO线程以及slave SQL线程，进行真正的复制工作。此外，该命令还可以指定终止条件，用来指定复制的结束条件。该命令可以选择性地开启slave IO线程或者slave SQL线程，或者两个线程全部开启，例如命令START SLAVE IO_THREAD仅仅开启slave IO线程，并不启动slave SQL线程；START SLAVE SQL_THREAD仅仅开启slave SQL线程，并不开启slave IO线程；而START SLAVE同时开启两个线程。

下面介绍START SLAVE命令的执行流程。

(1) 查看slave IO线程和slave SQL线程是否已经在运行了，如果都在运行，输出警告信息并结束执行。

(2) 查看Master_info的实例active_mi的信息是否完整。

(3) 如果携带了USER和PASSWORD参数，将其记录到active_mi中。

(4) 如果携带了终止条件，将其记录到active_mi->rli中。

(5) 启动还没有启动的slave线程。

8.5.5 STOP SLAVE命令——停止复制

该命令用于停止从库的复制工作。如果想在从库上执行备份或数据分析工作，可以执行该命令来暂时停止从库的更新。

执行STOP SLAVE命令，可以停止指定的slave线程，例如命令STOP SLAVE IO_THREAD仅仅停止slave IO线程，并不会停止slave SQL线程；STOP SLAVE SQL_THREAD仅仅停止slave SQL线程；STOP SLAVE将会停止两个线程。

该命令相对比较简单，主要执行流程如下。

(1) 检查是否有slave线程在运行，如果没有，输出警告结束，否则继续执行。
(2) 如果slave SQL线程在运行，发送信号给该线程，通知它结束执行。
(3) 如果slave IO线程在运行，发送信号给该线程，通知它结束执行。

8.5.6 slave IO线程

在执行START SLAVE命令开启复制功能时，该命令会通过调用start_slave_threads函数来启动slave IO线程和slave SQL线程。本节中，我们将分析slave IO线程的具体工作流程，来进一步加深对slave IO线程的理解。

slave IO线程的入口函数是void *handle_slave_io(void *arg)。下面我们给出该函数的主要代码，其中仅仅列出了关键的函数调用，忽略了一些细节，但并不影响整个理解过程。handle_slave_io(void *arg)的参数arg传入的是类Master_info的实例active_mi：

```
// sql/rpl_slave.cc

1. void *handle_slave_io(void *arg) {
2.     THD *thd= NULL;
3.     MYSQL *mysql;
4.     Master_info *mi = (Master_info*)arg;
5.     Relay_log_info *rli= mi->rli;
6.     int ret;

       // 初始化工作
7.     thd = new THD;
8.     init_slave_thread(thd, SLAVE_THD_IO);

9.     mi->slave_running = 1;
10.    mi->abort_slave = 0;

       // 连接到主库
11.    mi->mysql = mysql = mysql_init(NULL);
12.    safe_connect(thd, mysql, mi);

       // 获取主库信息，并且将从库的信息在主库上注册
13.    ret = get_master_version_and_clock(mysql, mi);
14.    if (!ret) ret = get_master_uuid(mysql, mi);
15.    if (!ret) ret = io_thread_init_commands(mysql, mi);
16.    if (ret) goto err;
17.    register_slave_on_master(mysql, mi);

       // 读取binlog事件，并且将其写入到relay-log中
18.    while (!io_slave_killed(thd,mi)) {
19.        request_dump(thd, mysql, mi);
```

```
20.         const char *evnet_buf;
21.         while (!io_slave_killed(thd,mi)) {
22.             ulong event_len = read_event(mysql, mi);
23.             if (event_len == packet_error) goto err;
24.             event_buf = (const char*)mysql->net.read_pos + 1;
25.             queue_event(mi, event_buf, event_len);

26.             flush_master_info(mi, FALSE);
27.             if (rli->log_space_limit && rli->log_space_limit < rli->log_space_total)
28.                 wait_for_relay_log_space(rli);
29.         }
30.     }
31.}
```

第7行到第10行代码用于进行一些初始化工作，包括为当前线程创建一个THD的实例，用于保存当前线程的上下文信息，并且将mi->slave_running设置为1，表示slave IO线程正在运行。

第11行和第12行代码在初始化完成之后，接着调用safe_connect函数建立到主库的连接。函数返回成功后，参数mysql就代表和主库的连接，后面所有和主库的通信都是通过该连接进行的。

第13行代码调用get_master_version_and_clock函数获取主库的版本号、主库的系统时间以及时区等信息，同时获取主库的server_id。获取了主库的系统时间和时区信息后，就可以计算出主库和从库的时间偏差，这对延迟复制是非常有用的。

第14行代码调用get_master_uuid函数获取主库的uuid信息，并将其记录下来。

第15行代码调用io_thread_init_commands函数将从库的uuid告诉主库。io_thread_init_commands函数通过在当前连接上执行SET @slave_uuid=uuid命令将从库的uuid信息存储到当前连接的用户变量中，这样主库就可以通过获取当前连接中用户变量@slave_uuid的值来得知从库的uuid信息。

第17行代码调用register_slave_on_master函数向主库发送COM_REGISTER_SLAVE命令，将从库在主库上进行注册。

第19行代码调用request_dump函数，告知主库应该从何处开始发送binlog事件。通常情况下，request_dump函数发送COM_BINLOG_DUMP命令给主库，而在GTID模式下，会发送COM_BINLOG_DUMP_GTID命令给主库，后续我们会继续介绍相关的内容。

第21行到第29行代码说明，slave IO线程进入一个while循环，用于进行事件的读取和记录。read_event从连接中读取事件，如果没有事件，将会阻塞在这里等待主库发送事件，而queue_event将读取的事件写入到relay-log文件中。flush_master_info将Master_info实例mi的信息同步到master.info文件中。

queue_event函数的主要工作是将收到的事件写入到relay-log中，其代码如下所示：

```
// sql/rpl_slave.cc

1. int queue_event(Master_info *mi,const char *buf, ulong event_len) {
2.     Log_event_type event_type= (Log_event_type)buf[EVENT_TYPE_OFFSET];
3.     mysql_mutex_t *log_lock= rli->relay_log.get_log_lock();
4.     int inc_pos = 0;

5.     switch (event_type) {
6.     case ROTATE_EVENT:
           // 更新 master_log_file和master_log_pos 信息
7.         ...
8.        break;
9.     case HEARTBEAT_LOG_EVENT:
           // 更新心跳时间
10.        ...
11.        goto skip_relay_logging;
12.        break;
13.        ...
14.    default:
15.        inc_pos= event_len;
16.        break;
17.    }

       //将事件写入到 relay-log
18.    mysql_mutex_lock(log_lock);
19.    rli->relay_log.append_buffer(buf, event_len, mi);
20.    mi->set_master_log_pos(mi->get_master_log_pos() + inc_pos);
21.    mysql_mutex_unlock(log_lock);

22.skip_relay_logging:
23.err:
24.    ...
25.}
```

第5行到第17行代码根据事件的类型做不同的处理。例如，当接收到的是心跳事件时，更新最近心跳时间，然后忽略该事件；当我们接收到一个类型为ROTATE_EVENT的事件时，说明主库的binlog发生了切换，我们需要更新Master_info实例mi中master_log_file和master_log_pos的信息来反映这种变化。

第18行和第19行代码用于获取relay-log的锁，然后调用append_buffer将事件写入到relay-log中。append_buffer函数主要完成了以下功能。

(1) 调用my_b_append函数将接收到的事件写入到relay-log的缓存中。

(2) 调用flush_and_sync函数，将io_cache中的内容刷新到relay-log并持久化到磁盘。

(3) 在将事件写入到relay-log文件中之后，判断当前relay-log文件的大小是否达到了max_relay_log_size指定的大小，如果大于就需要进行relay-log的切换。

(4) 这时slave SQL线程还不知道relay-log有更新了，所以需要调用signal_update函数通知slave SQL线程relay-log有更新了，可以继续进行重放动作了。

第20行代码中，由于我们读取了主库的事件，所以需要更新Master_info实例mi中master_log_file和master_log_pos的信息。

slave IO线程的大体执行流程是这样的，其中省略了一些细节，例如和主库的连接断开了之后的重连等流程，但这并不影响我们理解它的工作原理。有兴趣的读者可以自行翻阅该部分源代码。

8.5.7　slave SQL线程

slave SQL线程的入口函数是void *handle_slave_sql(void *arg)，传入的参数arg和slave IO线程一样，是Master_info的实例active_mi。下面我们给出该函数的关键代码：

```
// sql/rpl_slave.cc

1. void *handle_slave_sql(void *arg) {
2.     THD *thd;
3.     Relay_log_info *rli = ((Master_info*)arg)->rli;

       // 初始化工作
4.     thd = new THD;
5.     rli->slave_running = 1;
6.     init_slave_thread(thd, SLAVE_THD_SQL);

7.     rli->init_relay_log_pos(rli->get_group_relay_log_name(),
           rli->get_group_relay_log_pos());

           // 检查util条件
8.     ...

9.     while (!sql_slave_killed(thd,rli)) {
           // 检查util条件
10.         ...

           // 读取relay-log的事件并且重放
11.        Log_event *ev = next_event(rli);

12.        apply_event_and_update_pos(ev, thd, rli...);
13.        ...
14.    }
15.    ...
16.}
```

第4行到第6行代码用于进行一些初始化工作。将rli->slave_running设置为1，表示slave SQL

线程正在运行。

第7行代码调用init_relay_log_pos函数定位到上次重放的位置。

第9行到第14行代码是一个循环，不断从relay-log中读取事件然后进行重放。

第11行代码调用next_event函数从relay-log读取下一个事件。如果成功读取事件，则继续执行。如果返回的是EOF，说明relay-log中没有新的事件，阻塞在这等待slave IO线程通知relay-log有更新。

第12行代码调用apply_event_and_update_pos函数重放事件，并且更新相关的位置信息。

8.5.8 master dump线程

前面介绍过，当从库调用START SLAVE命令启动复制功能时，从库的slave IO线程会建立一个到主库的连接，主库和从库之间通过该连接进行通信和后续binlog的传输工作。主库在服务端会分配相应的线程来处理该连接上的所有请求，该线程就是master dump线程。对于主库来说，slave IO线程就是一个普通的客户端。

在slave IO线程连接到主库之后，会获取主库的时间、server_id等信息，并且发送COM_REGISTER_SLAVE命令，将从库在主库上进行注册，最后发送COM_BINLOG_DUMP命令给主库，请求主库发送binlog事件。

下面给出了COM_BINLOG_DUMP命令的处理函数com_binlog_dump的源代码，这也是master dump线程最主要的工作：

```
// sql/rpl_master.cc

1. bool com_binlog_dump(THD *thd, char *packet, uint packet_length)
2. {
3.     ulong pos;
4.     String slave_uuid;
5.     const uchar *packet_position= (uchar *) packet;
6.     uint packet_bytes_todo= packet_length;

7.     status_var_increment(thd->status_var.com_other);
8.     thd->enable_slow_log= opt_log_slow_admin_statements;
9.     if (check_global_access(thd, REPL_SLAVE_ACL))
10.        DBUG_RETURN(false);

       // 从packet中读取position，position指明了从哪儿开始复制
11.    READ_INT(pos, 4);
12.    SKIP(2); /* flags 字段暂时没有使用*/
13.    READ_INT(thd->server_id, 4);
```

```
        // 获取slave的uuid，关闭该slave对应的残留的dump线程
14.     get_slave_uuid(thd, &slave_uuid);
15.     kill_zombie_dump_threads(&slave_uuid);

        // 调用mysql_binlog_send函数发送binlog
16.     general_log_print(thd, thd->get_command(), "Log: '%s'  Pos: %ld",
            packet + 10, (long) pos);
17.     mysql_binlog_send(thd, thd->strdup(packet + 10), (my_off_t) pos, NULL);

18.     unregister_slave(thd, true, true/*need_lock_slave_list=true*/);
19.     DBUG_RETURN(true);

20.error_malformed_packet:
21.     my_error(ER_MALFORMED_PACKET, MYF(0));
22.     DBUG_RETURN(true);
23.}
```

下面我们来分析下com_binlog_dump函数的执行流程。

第9行代码检查当前连接是否具有复制权限，这对应于我们创建复制账号时是否赋予了该账号REPLICATION SLAVE权限。如果没有复制权限，直接结束运行，否则继续运行。

第11行到第13行代码用于从packet中读取发送binlog事件的开始位置，获取从库的server_id信息。

第14行代码调用get_slave_uuid函数获取slave的uuid信息。前面介绍过，从库的slave IO线程在连接到主库之后会将自己的uuid信息通过SET @slave_uuid=uuid命令保存到当前连接的用户变量@slave_uuid中，所以get_slave_uuid函数只需要取出当前连接的用户变量@slave_uuid的值就可以了。

第15行代码调用kill_zombie_dump_threads检查该从库之前是否有对应的master dump线程还没有关闭，如果有，就将其关闭。master dump线程在发送完binlog事件后会进入等待状态，等待binlog更新，如果在此期间slave停止了，然后又快速重新连接到该主库，那么在主库上会出现两个master dump线程与该slave对应：一个是之前建立的，处于等待binlog更新的master dump线程；另一个是由于从库最新连接上来而创建的master dump线程。此时，之前建立的那个master dump线程需要关闭。

第17行代码执行mysql_binlog_send函数向从库发送binlog事件。

第18行代码将从库的信息注销掉。如果执行到这里，说明已经从mysql_binlog_send函数返回，也就是说复制工作结束。

下面我们继续看一下binlog发送函数mysql_binlog_send的主要工作：

```
// sql/rpl_master.cc

1. void mysql_binlog_send(THD *thd, char *log_ident, my_off_t pos, const Gtid_set *slave_gtid_executed)
2.
3. {
     // 如果binlog没有打开或者server_id没有配置，则结束
4.     ...

     // 打开binlog文件，定位到正确的位置
5.     ...

6.     while (!net->error && net->vio != 0 && !thd->killed)
7.     {
8.         ...
9.         while (!(error= Log_event::read_log_event(&log, packet, log_lock,
                                                  current_checksum_alg,
                                                  log_file_name,
                                                  &is_active_binlog)))
10.        {
11.            ...
12.            my_net_write(net, (uchar*) packet->ptr(), packet->length())
13.            ...
14.        }

           // 进入阻塞状态，等待binlog更新
15.        ...
16.    }
17.}
```

先来看一下mysql_binlog_send函数的参数：参数thd表示该master dump线程的上下文信息；参数log_ident和pos指出发送binlog事件的开始位置；参数slave_gtid_executed表示slave已经执行过的GTID，不需要再发送给slave，该参数在开启GTID模式下的复制功能时使用。

下面给出了mysql_binlog_send函数的执行流程。

master dump线程的主要工作就是循环读取并发送binlog事件给从库。

第9行到第14行的循环调用Log_event::read_log_event函数读取binlog中的事件，调用my_net_write函数将事件发送给从库。

如果代码执行到第15行，说明读取到了EOF标志，表明所有binlog事件已经发送完毕，master dump线程进入阻塞状态，等待binlog的更新，并且定期地发送心跳包给从库。

通过以上的介绍，我们已经大体知道了master dump线程的执行过程，该部分的代码在sql/rpl_master.cc中，有兴趣想了解具体细节的读者可以自行翻阅。

8

8.6　半同步复制

默认情况下，复制是异步进行的，客户端提交事务给主库，主库将事务写入到存储引擎和binlog中后立刻返回给客户端，告诉客户端事务成功执行了，但此时该事务可能还没来得及复制到从库上。如果主库在此时发生崩溃或者主库所在的机器发生宕机，会导致主从切换的发生，此时客户端访问新选举出的主库时，是不会看到刚刚提交的数据的，但对于客户端来说，这些数据是提交成功了的。

8.6.1　半同步复制的工作原理

从MySQL 5.5以及MariaDB 5.5开始，MariaDB/MySQL通过插件的形式支持半同步复制。主库在执行完客户端提交的事务之后不是立刻返回给客户端，而是等待至少一个从库接收到了该事务之后才返回给客户端。

前面提到过，默认情况下，复制是异步进行的，主库将事务提交到存储引擎和binlog之后立刻返回给客户端，这里没有任何机制保证该事务同步到从库上。全同步的复制是指当主库执行完一个事务，所有的从库也执行了该事务之后才能返回给客户端。因为需要等待所有从库执行完该事务后才能返回，所以全同步复制的性能势必会受到严重的影响。半同步复制是异步复制和全同步复制的一个折中。相对于异步复制，半同步复制提高了数据的安全性，当客户端提交的事务返回之后，可以保证至少两个地方存在该事务，同时它也造成了一定的延迟，这个延迟至少是一个TCP/IP往返的时间。因此，半同步复制最好在低延时的网络中使用，例如局域网内，这样就能够缩小半同步复制的负面影响，否则，如果主库和从库之间的网络状况比较糟糕，半同步复制产生的事务提交延迟将是不可忍受的。后面我们会介绍几种在高延迟环境下使用半同步复制的思路，因为有的时候这种需求确实存在。例如，为了防止机房断电而导致的服务不可用，我们会进行跨机房容灾工作，将主库和从库放在不同的机房，而不同机房之间的网络延时通常是几毫秒甚至几十毫秒。半同步复制的主要流程如图8-3所示。

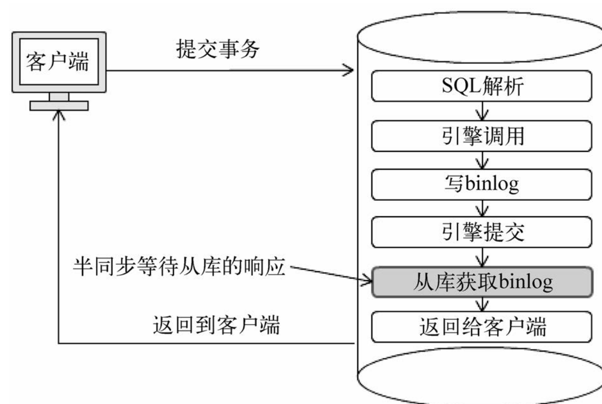


图8-3　半同步复制原理

8.6.2 半同步的安装和配置

本节中，我们将详细介绍半同步复制的安装以及配置。要想使用半同步复制，必须满足以下几个条件。

- MySQL 5.5/MariaDB 5.5或者以上的版本。
- has_dynamic_loading系统变量设置为YES。
- 复制已经配置了并且正在运行。

半同步复制是通过插件形式来实现的，所以在使用之前必须安装该插件。MariaDB/MySQL的半同步插件包含在发布版本中，解压发布版本，将semisync_master*文件复制到主库的plugins文件夹中，将semisync_slave*文件复制到所有从库的plugins文件夹中。当然，你也可以使用MariaDB/MySQL的源代码进行编译，获得半同步复制插件：

```
jinpengzhang@jinpengzhang:/usr/local/mysql/lib/plugin$ ls
adt_null.so  auth_socket.so       daemon_example.ini    mypluglib.so       qa_auth_interface.so
semisync_master.so  validate_password.so
auth.so      auth_test_plugin.so  libdaemon_example.so  qa_auth_client.so  qa_auth_server.so
semisync_slave.so
```

接下来分别在主库和从库上执行INSTALL PLUGIN命令加载半同步插件。

在主库上加载半同步复制插件的代码如下：

```
mysql> INSTALL PLUGIN rpl_semi_sync_master SONAME "semisync_master.so";
```

在从库上加载半同步复制插件的代码如下：

```
mysql> INSTALL PLUGIN rpl_semi_sync_slave SONAME "semisync_slave.so";
```

此时可以在主库和从库上执行SHOW PLUGINS命令来查看插件是否安装成功。

在从库上查看插件的安装情况：

```
mysql> SHOW PLUGINS;
+-----------------------+----------------- +----------------+------------------+--------+
| Name                  | Status           | Type           | Library          | ...    |
+-----------------------+------------------+----------------+------------------+--------+
| binlog                | ACTIVE           | STORAGE ENGINE | NULL             | ...    |
...
| rpl_semi_sync_slave   | ACTIVE           | REPLICATION    | semisync_slave.so | ...    |
+-----------------------+------------------+----------------+------------------+--------+
```

在主库上查看插件的安装情况：

```
mysql> SHOW PLUGINS;
+---------------------+----------+----------------+-------------------+--------+
| Name                | Status   | Type           | Library           | ...    |
+---------------------+----------+----------------+-------------------+--------+
| binlog              | ACTIVE   | STORAGE ENGINE | NULL              | ...    |
...
| rpl_semi_sync_master | ACTIVE  | REPLICATION    | semisync_master.so | ...   |
+---------------------+----------+----------------+-------------------+--------+
```

插件安装完之后，接下来必须在主库和从库上都开启插件，如果只在一端开启插件，复制还是异步进行的。可以通过设置主库和从库上的相应变量来开启插件。

在主库上开启半同步复制插件：

```
mysql> SET GLOBAL rpl_semi_sync_master_enabled = 1;
mysql> SET GLOBAL rpl_semi_sync_master_timeout = N; // 单位为毫秒
```

在从库上开启半同步复制插件：

```
mysql> SET GLOBAL rpl_semi_sync_slave_enabled = 1;
```

变量rpl_semi_sync_master_timeout表示主库等待从库响应的超时时间，如果在这个时间内还没有收到从库的响应，复制将切换到异步。

至此，我们已经配置好了半同步复制插件所需的相关条件，但是此时的复制仍然是异步的，这是因为我们是在复制正在进行的状态下配置的半同步复制。想要使半同步复制生效，必须先停止slave IO线程，然后重新开启该线程，这样slave IO线程在连接到主库之后，会以半同步的方式在主库上注册，之后半同步复制才能生效。

在从库上执行以下命令来重启slave IO线程：

```
mysql> STOP SLAVE IO_THREAD;
mysql> START SLAVE IO_THREAD;
```

通过设置全局变量来开启半同步复制的方式在MariaDB/MySQL重新启动之后，变量的值又会变回到默认值，所以必须在每次MariaDB/MySQL启动之后都要设置变量的值，如果忘记设置了，复制默认还是异步的。为了避免这种情况发生，可以在MariaDB/MySQL启动之前将这些值写入到配置文件中，具体如下所示。

主库配置文件：

```
[mysqld]
rpl_semi_sync_master_enabled=1
rpl_semi_sync_master_timeout=10000 # 10 秒
```

从库配置文件：

```
[mysqld]
rpl_semi_sync_slave_enabled=1
```

8.6.3 半同步复制的实现

前面已经介绍了半同步复制的工作流程，本节中我们将从源代码的角度来分析半同步复制是如何实现的。半同步复制是由主库和从库协同完成的，这里我们将从主库和从库两个角度分别展开讨论。在MySQL/MariaDB中，如果开启的是每连接每线程模式，每个客户端连接到服务端之后，都会在服务端生成一个线程处理该客户端的所有请求，我们称该服务端线程为**事务线程**。

1. 主库上的工作

在主库上，半同步复制的主要工作由类ReplSemi SyncMaster来完成。

- 类ReplSemiSyncMaster

我们先来看一下类ReplSemiSyncMaster的定义：

```
// plugin/semisync/semisync_master.h

Class ReplSemiSyncMaster
{
private:
    ActiveTranx *active_tranxs_;              /* 正在等待从库响应的活跃事务链表 */
    bool init_done_;                          /* 对象是否初始化 */

    mysql_cond_t COND_binlog_send_;           /* 用于事务线程与dump线程之间的通知 */
    mysql_mutex_t LOCK_binlog_;               /* 多线程修改对象成员之前必须获取该锁 */

    /* 从库已经响应的最大位置 */
    Bool reply_file_name_inited_;
    Char reply_file_name_[FN_REFLEN];
    my_off_t reply_file_pos_;

    /* 所有事务线程中等待从库响应的最小位置 */
    bool wait_file_name_inited_;
    char wait_file_name_[FN_REFLEN];
    my_off_t wait_file_pos_;

    /* binlog的最大位置 */
    Bool commit_file_name_inited_;
    Char commit_file_name_[FN_REFLEN];
    my_off_t commit_file_pos_;

    volatile bool master_enabled_;            /* 对应于rpl_semi_sync_master_enabled变量 */
    Unsigned long wait_timeout_;              /* 对应于rpl_semi_sync_master_timeout变量 */
    Bool state_;                              /* 当半同步超时的时候会暂时关闭半同步 */
```

```
private:
    bool is_on() const { return (state_); }
    void cond_broadcast();
    int  cond_timewait(struct timespec *wait_time);

    void set_master_enabled(bool enabled) { master_enabled_ = enabled; }

    int switch_off();
    int try_switch_on(int server_id, const char *log_file_name, my_off_t log_file_pos);

public:
    int enableMaster();                        /* 关闭主库半同步 */
    int disableMaster();                       /* 开启主库半同步 */
    bool getMasterEnabled() const { return master_enabled_; }

    int commitTrx(const char *trx_wait_binlog_name,my_off_t trx_wait_binlog_pos);
    int writeTranxInBinlog(const char *log_file_name,my_off_t log_file_pos);
    int readSlaveReply(NET *net, uint32 server_id, const char *event_buf);
    int reportReplyBinlog(uint32 server_id,const char *log_file_name,
    my_off_t end_offset,bool skipped_event= false);
};
```

在继续介绍半同步复制的实现之前，我们需要清楚binlog是线性的，这对我们理解半同步复制中的各种位置关系非常有帮助。如图8-4所示，“已响应的最大位置”表示在此之前的所有binlog从库（至少一个从库）已经接收到，在“已响应的最大位置”之前的事务是不需要等待的，可以直接返回给客户端，例如图8-4中的事务T2，而在“已响应的最大位置”之后的所有事务必须等待，例如图8-4中的事务T1和事务T3。当收到一个从库的响应中包含的位置大于“已响应的最大位置”时，“已响应的最大位置”就应该相应地往后移动。

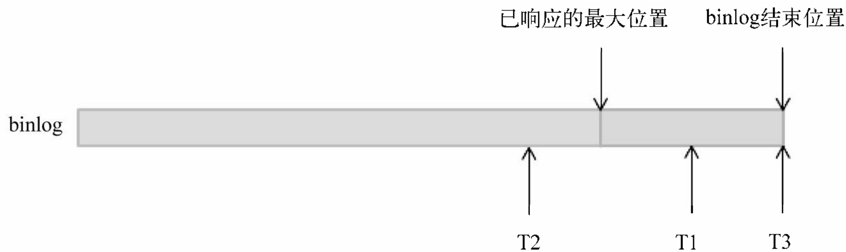


图8-4　半同步复制中的位置关系

接着我们给出类ReplSemiSyncMaster中各个成员的含义。

- active_tranxs_：该成员用于管理所有正在等待从库响应的事务。事务在binlog中有个结束位置，该结束位置以一个结构体TranxNode表示。当有多个事务在等待从库的响应时，所有事务的TranxNode按照事务在binlog中结束位置的前后关系排列，用成员next_组织起来，形

成一个链表，保存在active_tranxs_中。例如，图8-4中的事务T1和事务T3都需要等待从库的响应，那么它们就会按顺序组成一个链表（T1, T3）。下面是TranxNode结构的定义：

```
// plugin/semisync/semisync_master.h

struct TranxNode {
    char      log_name_[FN_REFLEN];
    my_off_t  log_pos_;
    struct TranxNode *next_;
};
```

- COND_binlog_send_：负责master dump线程和事务线程之间的通知。当事务线程将事务提交到存储引擎和binlog之后，如果需要等待从库的响应，那么会阻塞在该条件变量上，等待master dump线程接收到从库的响应之后将其唤醒。
- reply_file_name_和reply_file_pos_：这两个成员一起标识了图8-4中的“已响应的最大位置”。也就是说，在此位置之前的所有事务从库确定已经接收到了。
- reply_file_name_inited_：如果该成员为true，表明reply_file_name_和reply_file_pos_的值有效。
- wait_file_name_和wait_file_pos_：这两个成员标识了所有正在等待从库响应的事务中“最小”的一个，这里最小的意思是在binlog中的结束位置最靠前，例如图8-4中的事务T1。
- commit_file_name_：当前binlog的文件名。
- commit_file_pos_：当前binlog的结束位置，用于检查wait_file_name_、wait_file_pos_、reply_file_name_和reply_file_pos_的合法性。

接着我们介绍一下类ReplSemiSyncMaster中几个成员函数的作用。

- commitTrx：事务线程将事务提交到存储引擎和binlog之后会调用该函数，进入等待状态，等待master dump线程通知该事务已经被从库接收了或者发生超时。
- writeTranxInBinlog：当事务被提交到binlog中之后，会调用该函数来更新commit_file_name_以及commit_file_pos_的值。
- readSlaveReply：master dump线程发送完一个事务的binlog事件之后，会调用该函数等待从库响应，接收到响应之后，会通知所有正在等待的事务线程。
- reportReplyBinlog：它被readSlaveReply函数调用，通知所有正在等待的事务线程接收到了从库的响应，而事务线程各自检查从库返回的响应包中包含的位置是否大于等于自己的事务在binlog中的结束位置，如果大于等于，说明从库已经接收到了该事务，可以返回给客户端了，否则需要继续等待下去。

类ReplSemiSyncMaster在全局只有一个实例repl_semisync，该实例被所有的master dump线程和事务线程共同使用。

- commitTrx函数

当事务线程将事务提交到存储引擎和binlog之后，如果开启半同步，会调用commitTrx函数等待master dump线程接收到从库的响应之后通知自己。下面我们列出了该函数的主要源代码：

```
// plugin/semisync/semisync_master.cc

1. int ReplSemiSyncMaster::commitTrx(const char *trx_wait_binlog_name,
                                  my_off_t trx_wait_binlog_pos)
2. {
3.     if (master_enabled_ && trx_wait_binlog_name)
4.     {
5.         struct timespec abstime;

6.         mysql_mutex_lock(&LOCK_binlog_);
7.         if (!master_enabled_ || !is_on()) goto l_end;

8.         gen_timeout_abstime(&abstime, wait_timeout_); /* 获取超时的绝对时间 */
9.         while (is_on())
10.        {
11.            if (reply_file_name_inited_)
12.            {
13.                int cmp = ActiveTranx::compare(reply_file_name_, reply_file_pos_,
                       trx_wait_binlog_name, trx_wait_binlog_pos);
14.            if (cmp >= 0) /* 已经发送了该事务所有相关的binlog到从库，不需要等待 */
15.                break;
16.            }

17.            if (wait_file_name_inited_)
18.            {
19.                int cmp = ActiveTranx::compare(trx_wait_binlog_name,
                       trx_wait_binlog_pos, wait_file_name_, wait_file_pos_);
20.                if (cmp <= 0)
21.                {
22.                    strcpy(wait_file_name_, trx_wait_binlog_name);
23.                    wait_file_pos_ = trx_wait_binlog_pos;
24.                }
25.            }
26.            else
27.            {
28.                strcpy(wait_file_name_, trx_wait_binlog_name);
29.                wait_file_pos_ = trx_wait_binlog_pos;
30.                wait_file_name_inited_ = true;
31.            }

32.            wait_result = cond_timewait(&abstime); /* 阻塞在此，等待超时或者dump线
                                                      程通知收到从库的响应 */
33.            if (wait_result != 0) /* 超时 */
34.            {
```

```
35.                 rpl_semi_sync_master_wait_timeouts++;
36.                 switch_off(); /* 关闭半同步，并唤醒所有等待的线程 */
37.             }

38.         }
39.     }
40.l_end:
41.     return 0;
42.}
```

该函数的参数trx_wait_binlog_name和trx_wait_binlog_pos表示该事务在binlog中的结束位置。

第3行代码判断半同步复制是否可用，如果不可用，该函数直接返回，否则进入半同步复制的流程。如果主库执行了SET GLOBAL rpl_semi_sync_master_enabled=1命令，那么变量master_enabled_的值等于1。

代码第6行获取锁。因为后面的代码可能会修改某些成员的值，而所有的master dump线程和事务线程共用一个ReplSemiSyncMaster实例，所以必须用锁保护。该锁会在第32行代码中的cond_timewait函数内部释放。

第7行代码用于判断半同步开关是否打开。当半同步复制发生超时时，会暂时关闭半同步复制（第33行到第37行代码）。is_on函数用于判断半同步复制是否因为超时而被关闭。

第8行代码生成了一个该事务等待响应的绝对超时时间abstime，在第32行调用cond_timewait时会使用该变量。

接下来进入一个while循环，进行真正的工作。如果半同步被暂时关闭，那么事务线程直接返回给客户端。

第11行到第16行代码用于判断当前“已响应的最大位置”是否大于等于事务在binlog中的结束位置，如果大于等于，表明该事务的所有binlog事件已经被从库接收，可以结束该函数，事务线程可以返回给客户端。

由于binlog是排他写入的，并且事务是一次性完整写入的，所以不同事务在binlog中的位置取决于该事务是何时写入binlog中的。第17行到第31行代码做的事情是记录所有等待事务中最先写入binlog的事务在binlog中的结束位置，我们称该位置为**等待的最小位置**。

第32行代码调用了cond_timewait函数，事务线程将会阻塞在这，等待超时或者被master dump线程在接收到从库发来的响应之后唤醒。

第33行代码判断cond_timewait是否因为超时而返回的，如果是，则调用switch_off函数暂时关闭半同步复制并且唤醒所有正在等待的事务线程，避免无限等待下去，使数据库进入假死状态。如果事务线程是因为master dump线程接收到从库的响应，并且更新了“已响应的最大位置”而

被唤醒的，这时程序会跳到第11行，继续判断最新的“已响应的最大位置”是否大于等于事务在binlog中的结束位置。

- master dump线程在半同步中的工作

master dump线程发送完一个事务的所有事件之后，会调用readSlaveReply函数来等待从库的响应。当收到从库的响应之后，解析其中的位置信息，然后调用reportReplyBinlog函数，根据情况来判断是否需要唤醒所有正在等待的事务线程。reportReplyBinlog函数的主要代码如下所示：

```
// plugin/semisync/semisync_master.cc

1. int ReplSemiSyncMaster::reportReplyBinlog(uint32 server_id,
                                              const char *log_file_name,
                                              my_off_t log_file_pos)
2. {
3.     int  cmp;
4.     bool  need_copy_send_pos = true;
5.     bool  can_release_threads = false;

6.     if (!(getMasterEnabled())) /* 判断半同步是否开启 */
7.         return 0;

8.     mysql_mutex_lock(&LOCK_binlog_);

9.     if (!getMasterEnabled()) /* 判断半同步是否开启 */
10.        goto l_end;

11.    if (!is_on()) /* 如果半同步由于超时暂时关闭，尝试重新开启 */
12.        try_switch_on(server_id, log_file_name, log_file_pos);

13.    if (reply_file_name_inited_)
14.    {
15.        cmp = ActiveTranx::compare(log_file_name, log_file_pos,
                                     reply_file_name_, reply_file_pos_);
16.        if (cmp < 0)
17.            need_copy_send_pos = false;
18.    }

19.    if (need_copy_send_pos)
20.    {
21.        strcpy(reply_file_name_, log_file_name);
22.        reply_file_pos_ = log_file_pos;
23.        reply_file_name_inited_ = true;

24.        active_tranxs_->clear_active_tranx_nodes(log_file_name, log_file_pos);
25.    }

26.    cmp = ActiveTranx::compare(reply_file_name_, reply_file_pos_,
```

```
                              wait_file_name_, wait_file_pos_);
27.    if (cmp >= 0)
28.    {
           /* 至少一个事务线程可以返回 */
29.        can_release_threads = true;
30.        wait_file_name_inited_ = false;
31.    }

32.l_end:
33.    mysql_mutex_unlock(&LOCK_binlog_);

34.    if (can_release_threads) /* 唤醒所有事务线程 */
35.        cond_broadcast();

36.    return 0;
37.}
```

先来看一下reportReplyBinlog函数的参数:server_id表明是哪个从库发来的响应;log_file_name和log_file_pos是包含在响应中的位置信息,说明在此位置之前的所有binlog事件在该从库中都接收到。

第6行代码用于判断半同步复制是否开启，如果没有开启直接返回，否则继续执行。

第8行到第9行代码在获取锁保护之后又做了一次判断，判断半同步是否开启。

第11行到第12行代码用于检查半同步复制是否由于超时而被关闭,如果是的话尝试重新开启半同步复制。由此可知，当半同步复制发生超时时，会暂时关闭半同步复制，在此期间复制是异步进行的，当再次接收到从库的响应时半同步复制才会重新开启。

第13行到第18行代码用于判断接收到的响应中包含的位置信息是否比当前“已响应的最大位置”更靠后，如果是的话，需要更新“已ACK的最大位置”。

第19行到第25行代码说明，如果当前接收到的响应中包含的位置比“已响应的最大位置”更靠后，更新“已响应的最大位置”，并且将所有位于该位置之前的正在等待的事务从活跃事务列表中删除，表示这些事务不需要继续等待下去了。

第26行到第31行代码用于判断是否有事务线程需要被唤醒。

第34行和第35行代码说明，如果有事务线程需要被唤醒，唤醒所有等待的事务线程，让它们各自检查自己是否需要继续等待下去。这时所有被唤醒的事务线程会执行commitTrx函数的第11行到第16行，判断是否可以返回给客户端，还是需要继续等待下去。

2. 从库上的工作

下面我们以从库的角度来看一下slave IO线程在半同步复制开启的情况下做了哪些额外的工作。

在从库上，slave IO线程负责从主库接收binlog事件，并将其写入到relay-log中。当半同步复制开启的情况下，接收完一个事务的所有binlog事件之后，slave IO线程会向主库发送一个响应，告知主库从库已经接收到了该事务。

- ReplSemiSyncSlave类

在从库上，控制半同步复制的是ReplSemiSyncSlave类，该类在全局有唯一的实例repl_semisync被slave IO线程所使用。由于在MySQL中一个从库只能有一个主库，所以只会有一个slave IO线程，不存在多个线程同时操作该类的情况，因此，相对于ReplSemiSyncMaster类，ReplSemiSyncSlave类少了很多并发控制。

在目前版本的MariaDB中，如果使用多源复制，那么半同步复制是不能使用的，导致这个问题的原因与半同步复制插件中类ReplSemiSyncSlave的实现有关。

下面我们给出类ReplSemiSyncSlave的定义：

```
// plugin/semisync/semisync_slave.h

class ReplSemiSyncSlave
{
private:
    bool init_done_;                     /* 实例是否初始化 */
    bool slave_enabled_;                 /* 半同步是否开启 */
    MYSQL* mysql_reply;                  /* 用于向主库发送响应 */

public:
    ReplSemiSyncSlave() :slave_enabled_(false) {}
    ~ReplSemiSyncSlave() {}

    void initObject();
    bool getSlaveEnabled() const { return slave_enabled_; }
    void setSlaveEnabled(bool enabled) { slave_enabled_ = enabled; }

    /* 当接收完一个事务的所有事件之后，会调用该函数向主库发送响应包 */
    int slaveReply(MYSQL *mysql, const char *binlog_filename,my_off_t binlog_filepos);
};
```

类ReplSemiSyncSlave的结构非常简单，下面我们给出这个类各个成员的具体含义。

- init_done_：如果为true，表示ReplSemiSyncSlave实例已经初始化。
- slave_enabled_：标志半同步是否开启。
- mysql_reply：和主库的连接，用于向主库发送响应。

下面简要介绍一下ReplSemiSyncSlave类中各个成员函数的具体作用。

- initObject：用于初始化ReplSemiSyncSlave实例。

- ❑ getSlaveEnabled：获取半同步开启状态。
- ❑ setSlaveEnabled：设置半同步开启状态。
- ❑ slaveReply：当接收完一个事务的所有binlog事件之后，会调用该函数向主库发送响应。

● slave IO线程在半同步复制中的额外工作

下面我们看一下从库的slave IO线程从启动到退出，在半同步复制中做了哪些额外的工作。

(1) slave IO线程在发送COM_BINLOG_DUMP命令请求主库发送binlog事件之前，会询问主库是否支持半同步复制，如果不支持，就算从库支持半同步复制，也会将半同步复制关闭。

(2) 如果第(1)步中主库给出的答复是支持半同步复制，接下来从库会发送SET @rpl_semi_sync_slave= 1命令给主库，即设置该用户变量的值为1，其含义是至少有一个从库使用半同步复制。当主库的master dump线程发送完一个事务的所有事件之后，会检查用户变量@rpl_semi_sync_slave的值，来确定是否需要等待从库响应。

(3) 从库接收完一个事务的所有binlog事件之后，会调用ReplSemiSyncSlave的slaveReply函数向主库发送包含位置信息的ACK确认。

8.6.4 半同步复制的变种

通常情况下，半同步复制是保证数据安全性很好的解决方案，然而在一些特殊的场景或者需求下，普通的半同步复制不能满足需求。本节中，我们将介绍为了满足特殊需求而存在的两个半同步复制的变种。

1. group-ack半同步复制

由于半同步复制需要等待从库发送响应，为了减少半同步复制的额外开销，通常情况下我们会在低延时的网络情况下使用半同步复制，例如主库和从库位于同一个局域网内。但在一些特殊的情况下，例如为了防止由于机房断电而导致的数据不可访问，我们会做跨机房容灾，将主库和从库放在不同的机房。但通常不同的机房之间的网络延时要比局域网内的网络延时大得多，一般会达到几毫秒甚至几十毫秒。我们假设两个机房之间的网络延时为10ms，不考虑其他因素的影响，那么主库的吞吐率就可以粗略地估算为1000 / 10 = 100事务数/秒，这对数据库服务来说是比较低的。

我们知道，在TCP传输协议中，接收方是需要发送响应给发送端，告知对方自己已经接收到了数据包。发送端并不是每发送完一个数据包就等待接收端发送响应，这样会使TCP传输速度非常慢。发送端在第一个数据包还没有收到响应的情况下是可以接着发送第二个数据包的，只要未收到响应的数据包的个数小于滑动窗口的大小，就可以继续向接收端发送数据包。同时接收端也不是对每个数据包都进行响应，可以对多个数据包进行一次响应，例如接收端收到了序列号分别为1、2、3、4的4个数据包，那么接收端可以只发送一个4号包的响应就可以。

TCP数据包的顺序性和事务在binlog中的结束位置的顺序性非常相似，例如事务A在binlog中的结束位置是pos1，事务B在binlog中的结束位置是pos2，如果pos2大于pos1，当主库接收到从库发来的对pos2的响应时，也就是说从库接收到了事务B的所有事件，那么位于事务B之前的事务A的所有事件，从库肯定也接收完了。可以利用这种性质来对跨机房的半同步复制进行优化。并不是每发送完一个事务之后都去等待从库的响应，而是发送完一批事务之后再等待一个响应。这样可以将一次等待响应的时间开销平摊到多个事务上，减少了单个事务在等待从库响应上所花的时间。例如，发送10个事务之后master dump线程进入等待从库的响应包，假设主库和从库的延时为10ms，不考虑其他因素的影响，那么主库的吞吐率可以粗略地估算为1000 / 10 × 10 = 1000事务数/秒。相比于之前，这种方案可以显著提高数据库的吞吐率。

当然，group-ack半同步复制在真正的实践中需要考虑的问题还有很多，例如当数据库系统空闲的时候，一个group应该设置为多大。如果设置得太大，将会导致收集一个group的事务要花费很长的时间，这会导致长时间不能返回给客户端。group-ack只有在大量并发事务的时候才能很好地发挥作用。

2. wait N-ack半同步复制

通常的半同步复制是只要有一个从库返回了响应，主库就可以返回给客户端。当主库返回给客户端时，如果主库和刚刚返回响应的那个从库一起宕机，这时会发生主从切换，从其他的从库中选举一个从库作为主库，在新的主从复制结构中，客户端刚刚提交的那条事务是不存在的，客户端是不能访问刚刚提交的数据的。但这种情况出现的可能性非常低，通常情况下是不需要考虑的，但对于一些安全级别要求非常高的商业系统，确实需要考虑这种情况。

wait N-ack半同步复制的思路如下：不是只要一个从库返回响应就返回给客户端，而是等待N（$N > 1$）个从库返回响应之后才返回给客户端。当然，如果N越大，系统的吞吐率就越低，但安全性也越高，当N为从库的个数时，安全性最高，系统的吞吐率也是最低的，因为要等待所有的从库返回响应包才能返回给客户端。

8.6.5　半同步复制的潜在问题

半同步复制是为了解决在异步复制中，当从库落后主库时，主库宕机会发生某些数据不可访问的问题。但半同步复制也存在一些潜在的问题。

首先，不考虑其他因素的影响，半同步复制相对于异步复制系统的吞吐率会有所降低。当然，在网络状况很好的情况下，网络延时不是系统瓶颈的情况下，半同步复制可以达到和异步复制相同的吞吐率，但当网络状况不是很好时，可能会成为系统的瓶颈，所以我们推荐在网络状况好的情况下使用半同步复制，当然特殊场景或需求下除外。

其次，由于半同步复制在将事务提交到存储引擎和binlog之后需要等待从库响应，如果主库在等待的时候宕机了，这时这些事务还没来得及发送到从库上，客户端会收到事务提交失败的信

息，客户端会重新提交该事务到新选举的主库上，当宕机的主库重新启动，以从库的身份重新加入到该主从复制结构中时会发现，该事务在这个主从结构中被提交了两次，一次在之前的主库上，一次在新选举的主库上。

当主库在从库发送完响应包但是没有收到响应包这个期间宕机时，客户端会返回提交事务失败，但从库已经获取了该事务，这样客户端会重新提交这些事务到新选举的主库上，而此时该事务在这个主从复制结构中已经存在了。

虽然半同步复制存在以上提到的这些问题，但这些问题可以通过其他途径或者方案来避免。在通常情况下，半同步复制仍然是保证数据安全性的一种非常好的解决方案。

8.7 并行复制

在开启复制的情况下，从库的slave IO线程负责接收主库的binlog事件，然后将其写入到relay-log中，而slave SQL线程负责重放relay-log中的事件，这个过程在前面已经详细介绍过了。在使用复制的过程中，我们有时会发现从库的slave IO线程已经获取了主库的所有binlog事件，然而slave SQL线程重放的进度却跟不上slave IO线程拉取binlog事件的进度，导致从库落后主库，在主库上可以查询到的数据在从库上却查询不到。这可能有多种原因。例如，主库和从库的机器资源不对称，从库机器的性能没有主库的好，在业务峰值的时候主库可以应对峰值的压力，而从库的性能到了一个瓶颈，导致重放的速度跟不上主库的更新。当然，通常情况下我们应该避免发生这种情况，主库和从库最好使用性能对称的机器，这样当主库宕机之后，发生主从切换，从库也能应对业务峰值时的压力。从库落后还有一个可能的原因，那就是主库进行高并发的写入，而从库是通过slave SQL单线程进行重放的，也就是说在从库上同一时刻只有一个事件被重放，没有充分利用系统资源，跟不上主库的步伐。如果是由于这种原因而造成的从库落后，那么MySQL/MariaDB的并行复制功能就可以很好地发挥作用了。

8.7.1 MySQL的并行复制

MySQL在5.6.3版本中引入了并行复制功能，允许多个线程并发重放接收到的binlog事件。用户可以通过设置参数`slave_parallel_workers`来指定从库参与重放工作的线程的个数，默认情况下该参数的值为0，表明只有一个slave SQL线程负责重放，如果该参数不为0，从库除了会开启一个slave SQL线程外，还会额外开启参数指定数量的工作线程，slave SQL线程不直接参与重放，而是作为所有工作线程的协调器，负责将事件分发给工作线程，由工作线程负责重放。

8.7.2 MariaDB的并行复制

MariaDB在10.0.5中引入了并行复制的功能，允许从库的多个线程同时进行重放工作。你可以通过在配置文件中设置参数`slave_parallel_threads=n`来指定MariaDB创建多少个`worker`线程

来进行事件的重放工作。如果n为0，表示采用传统的复制模式——单个slave SQL线程负责所有的重放工作；如果n大于0，将会创建一个包含n个worker线程的线程池进行重放工作。

8.8　多源复制

MySQL能够轻松支持一主多从的复制结构，但是想将多个实例的数据复制到一个实例上还是比较难的，当然可以通过多级复制来达到目的，但这种方式不是很优雅，而且存在一定的问题，例如导致网络流量的增加以及产生冗余的binlog，等等。MariaDB很好地解决了这个问题。从10.0开始，MariaDB引入了多源复制的机制，允许一个从库可以指定多个主库作为数据源。多源复制允许将多个实例的数据进行聚合作为备份或者进行分析，如图8-5所示。

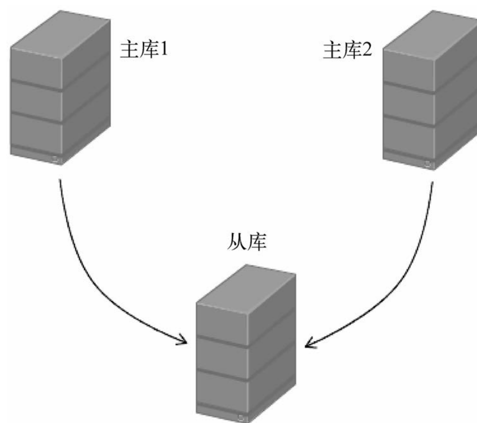


图8-5　MariaDB多源复制

8.8.1　多源复制的应用场景

多源复制的通常应用场景包括如下几种。

- 由于某些原因，数据被分片到多个数据库实例上，你想把这些数据聚集到一块进行数据分析等工作。
- 你有多个数据库实例，想把这些实例的数据使用一台机器进行备份。

当然，如果你想使用多源复制，必须了解它的一些限制。下面列出了使用多源复制需要注意的一些问题。

- 一个从库最多能有64个主库。通常情况下，64个主库已经足够多了，如果你的需求多于这个数，应该考虑下自己的设计是否合理。
- 每增加一个主库，在从库上都会创建两个线程（slave IO和slave SQL线程），也就是如果你的从库有N个主库，在多源复制开启的时候该从库上将会存在2N个slave线程。

- ❑ 每个主库必须配置不同的server_id。如果不这样做，当你想从从库把数据复制回主库时会出现问题。
- ❑ 建议不要开启innodb-recovery-update-relay-log选项，该选项对于InnoDB之外的存储引擎是不安全的。
- ❑ 目前版本（10.0.6）的MariaDB的多源复制会使半同步复制失效，MariaDB将来的版本会修复这个问题。

8.8.2 多源复制相关的命令

为了支持多源复制，MariaDB对复制中使用的一些命令做了语法扩展，同时引入了一些新的命令，下面我们一一介绍这些修改。

- ❑ CHANGE MASTER ["connection-name"] TO ...：为了在多个主库之间做区分，MariaDB允许在CHANGE MASTER TO命令执行时为从库到主库之间的连接进行命名，连接名的长度不能大于64个字符。在8.8.3节中，我们将会对该名称在MariaDB内部是如何起作用的做进一步的介绍。
- ❑ RESET SLAVE ["connection-name"]：类似于MySQL的RESET SLAVE命令，但是它可以通过指定连接名来重置指定主库的信息。
- ❑ SHOW RELAYLOG ["connection-name"] EVENTS：查看指定主库生成的relay-log的事件。在使用MariaDB的多源复制时，如果从库需要复制*N*个主库的数据，那么在从库上就会开启*N*对slave IO和slave SQL线程对，同时会生成*N*个relay-log。SHOW RELAYLOG EVENTS命令支持查看指定主库生成的relay-log。
- ❑ SHOW SLAVE ["connection-name"] STATUS：扩展了MySQL的SHOW SLAVE STATUS命令，可以查看指定主库的复制情况。
- ❑ SHOW ALL SLAVE STATUS：MariaDB针对多源复制新增的命令，使用该命令可以查看所有主库的复制情况。
- ❑ START SLAVE ["connection-name"]：启动指定主库的复制。
- ❑ START ALL SLAVE：MariaDB新增的命令。启动所有主库的复制工作，该命令会开启*N*对slave IO和slave SQL线程对来进行复制，*N*为主库的个数。
- ❑ STOP SLAVE ["connection-name"]：停止指定主库的复制工作。
- ❑ STOP ALL SLAVE：MariaDB新增的命令，用于停止所有的复制工作。
- ❑ default_master_connection选项：通过上面的介绍，我们看到，MariaDB通过连接名来区分不同的主库，许多复制相关的命令支持指定连接名来指定作用的主库。当我们在这些命令中不指定连接名时，会发生什么呢？MariaDB引入了default_master_connection选项，你可以在配置文件中或者在启动MariaDB的时候在命令行中指定该选项的值，那么如果你在执行复制命令时没有指定连接名，系统默认会使用default_master_connection参数的值。default_master_connection的默认值是空字符串。

8.8.3　MariaDB多源复制的实现

前面介绍MySQL的复制实现时，我们介绍了几个类的定义及其在复制中的作用，MariaDB的多源复制大体上和普通的复制是相同的，所以本节中我们并不打算把MariaDB多源复制相关的所有内容都列举出来，而是列举与MySQL复制有所区别的地方。

先来看一下用于保存主库信息的Master_info类，该类和MySQL相对应的Master_info类几乎一致，除了多了一个connection_name成员和一个cmp_connnection_name成员。这两个成员用于存储多源复制中的连接名，其中前者存储了原始的连接名，后者存储了连接名的小写，用于比较和查找操作。Master_info类的代码如下：

```
// sql/rpl_master.h

class Master_info
{
    ...
    LEX_STRING connection_name;          // 连接的名称
    LEX_STRING cmp_connection_name;      // 小写，用于查询和比较
    ...
};
```

在MySQL中，Master_info类全局只有一个实例（全局变量active_mi），因为MySQL只允许有一个主库，而MariaDB允许有多个主库，那么相应地就会存在Master_info的多个实例，既然有多个Master_info的实例，那么就需要对这些实例进行管理，类Master_info_index就是为了管理这些实例而存在的。下面我们给出该类的定义：

```
// sql/rpl_master.h

class Master_info_index
{
private:
    IO_CACHE index_file;
    char index_file_name[FN_REFLEN];               // 对应于multi-master.info文件

public:
    Master_info_index();
    ~Master_info_index();

    HASH master_info_hash;                         // 管理所有Master_info的实例

    bool init_all_master_info();                   // 读取所有master.info文件，初始化多源复制
    bool write_master_name_to_index_file(LEX_STRING *connection_name,bool do_sync);

    bool check_duplicate_master_info(LEX_STRING *connection_name, const char *host, uint port);
    bool add_master_info(Master_info *mi, bool write_to_file);
    bool remove_master_info(LEX_STRING *connection_name);
```

```
    // 通过连接的名字来找到Master_info的实例
    Master_info *get_master_info(LEX_STRING *connection_name,
                                 Sql_condition::enum_warning_level warning);

    bool start_all_slaves(THD *thd);
    bool stop_all_slaves(THD *thd);
};
```

接下来，我们给出类Master_info_index中各个成员的含义。

- index_file：读写multi_master.info文件的缓存。
- index_file_name：在MySQL的复制中，本地会生成一个master.info文件用于保存连接主库的信息以及slave IO线程接收主库binlog事件的进度情况；生成一个relay_log.info文件存储slave SQL线程重放工作的进度。在MariaDB的多源复制中，由于存在多个主库，针对每个主库都会生成一个单独的master.info文件，文件名为master-*connection_name*.info，其中*connection_name*为到主库的连接名。同时，MariaDB还生成了一个multi-master.info文件用于存储所有的连接名。成员index_file_name里保存的是multi-master.info文件的文件名。
- master_info_hash：这是一个哈希表，用于管理所有的Master_info实例，可以根据连接名来快速查询对应的Master_info实例。

下面列出了类Master_info_index中各个成员函数的含义。

- init_all_master_info：当MariaDB启动的时候，会调用init_all_master_info函数来从multi-master.info文件读取所有的连接名，然后对所有主库进行初始化。
- write_master_name_to_index_file：将连接名写入到multi-master.info文件。
- check_duplicate_master_info：在使用CHANGE MASTER TO命令时，会调用该函数来检查该主库是否已经存在。
- add_master_info：添加Master_info实例到master_info_hash。
- remove_master_info：从master_info_hash中删除Master_info实例。
- get_master_info：通过连接名从master_info_hash中获取对应的Master_info实例。
- start_all_slaves：该函数用于启动所有没有启动的复制线程，在执行START ALL SLAVES命令时被调用。
- stop_all_slaves：该函数用于停止所有的复制线程，在执行STOP ALL SLAVES命令时被调用。

8.9 GTID

GTID是Global Transaction Identifiers的缩写，表示全局事务id，其主要目的是为了简化复制。MySQL在5.6.5中引入了这个概念，MariaDB从10.0.2开始将这个概念引进来，支持基于GTID的复制。

8

本节中，我们首先介绍GTID的概念，然后介绍GTID在MySQL中是如何配置的及其在MySQL中的实现，最后对比下MariaDB中的GTID与MySQL中的不同之处，让读者对GTID有个全面深刻的了解。

8.9.1　GTID的概念

前面我们介绍过，在复制的过程中，从库通过记录主库的binlog文件名和偏移量来记录接收主库binlog事件的工作进度，下次开启复制的时候通过告知主库这些信息，让主库能够从正确的位置开始发送binlog事件给从库。基于GTID的复制不需要这些信息，在执行`CHANGE MASTER TO`命令时，也不需要指定`MASTER_LOG_FILE`和`MASTER_LOG_POS`参数，只需要指定`MASTER_AUTO_POSITION=1`就可以了，主库会根据从库发送过来的GTID集合信息来决定从何处开始发送binlog事件给从库以及发送哪些binlog事件给从库，这大大简化了数据库管理员在复制中的工作。

先来看一下GTID的概念。GTID是在数据库提交事务时创建的唯一标识符，该标识与事务是一一对应的关系。我们需要注意的是，它不仅在生成它的数据库上是唯一的，在一个主从复制结构内也是唯一的，这样才能唯一标识一个主从复制结构中的一条事务。GTID由两部分组成，如下所示：

```
GTID = source_id:transaction_id
```

`source_id`用于标识这个事务是在哪个数据库实例上产生的。在MySQL中，我们使用`uuid`作为`source_id`，`uuid`是由32个字符（`0-9`、`a-f`）以及4个-组成的字符串。`transaction_id`是一个序列号，取决于该事务在数据库上提交的顺序，该序列号从1开始。例如下面的GTID代表在`server_uuid`为`a194c9c6-566f-11e3-b67e-74867a2edb63`的数据库实例上提交的第6个事务：

```
a194c9c6-566f-11e3-b67e-74867a2edb63:6
```

接下来，我们介绍下GTID集合的概念。顾名思义，GTID集合就是由多个GTID组成的集合，一个GTID与一条事务相对应，一下GTID集合对应于一个事务集合。下面我们来看一个GTID集合：

```
a194c9c6-566f-11e3-b67e-74867a2edb63:1-6
```

这个集合表示在`server_uuid`为`a194c9c6-566f-11e3-b67e-74867a2edb63`的数据库实例上产生的事务号为1、2、3、4、5、6的6个事务。当然，上面的GTID集合是最简单的情况。GTID集合还能表示多个数据库实例上的事务集合，不同数据库实例之间用（,）隔开，同一数据库实例上的多个区间以冒号（:）隔开，例如：

```
a194c9c6-566f-11e3-b67e-74867a2edb63:1-6:8-23,0652420f-58bf-11e3-858f-74867a2edb63:1-8:17-25
```

8.9.2　在MySQL上配置基于GTID的复制

下面我们介绍一下在MySQL中如何开启基于GTID的复制，主要的配置过程分为以下几步。

(1) 同步主库和从库的数据。

(2) 停止主库和从库。

(3) 以--gtid_mode=on --log-bin --log-slave-updates --enforce-gtid-consistency模式启动主库和从库，在从库上需要添加--skip-slave-start选项，暂时不开启复制。

(4)在从库上执行CHANGE MASTER TO … MASTER_AUTO_POSITION = 1，使用基于GTID的自动位置功能替代MASTER_LOG_FILE和MASTER_LOG_POS参数。

(5)在从库上执行START SLAVE命令开启复制。

在gtid_mode=on的模式下，MySQL会在事务提交的时候生成一个GTID，然后在binlog中针对该事务额外记录一个Gtid事件：

```
mysql> show binlog events in "mysql-bin.000043";
+-------------    +------+--------------+-----------+------------+-------------+
| Log_name        | Pos  | Event_type    | Server_id | End_log_pos | Info         |
+-------------    +------+--------------+-----------+------------+-------------+
| mysql-bin.000043| 4    | Format_desc              | 1   | 120     | Server ver: 5.6.14
                                                                      -debug-log,    Binlog ver: 4 |
| mysql-bin.000015| 120  | Previous_gtids           | 2   | 231     | 0652420f-58bf-11e3-858f-
                                                                      74867a2edb63:1-2,
                                                                      a194c9c6-566f-11e3-b67e-
                                                                      74867a2edb63:1-10      |
| mysql-bin.000043| 191  | Gtid                     | 1   | 239     | SET @@SESSION.GTID_NEXT=
'                                                                     'a194c9c6-566f-11e3-b67e-
                                                                      74867a2edb63:11'      |
    ...
```

从上面的输出可以看到，在GTID开启的模式下，binlog会额外记录两种事件，这两种事情分别是Previous_gtids事件和Gtid事件。在事务开始之前，会记录一个Gtid事件与该事务相对应；事件Previous_gtids记录了该binlog文件之前执行过的所有事务对应的GTID集合，在系统启动时，MySQL读取该事件的内容来进行相应的初始化工作。

在下一节中，我们会再次分析这两个binlog事件。

8.9.3 MySQL中GTID的实现

介绍完了GTID、GTID集合的概念以及表示方法，下面我们从源代码的角度来分析GTID在MySQL内部是如何实现的以及在复制过程中GTID是如何工作的。

1. 相关的数据结构

首先，我们介绍几个实现GTID的核心数据结构。

- 类Sid_map

在MySQL中，我们使用uuid作为GTID中的source_id（第一部分）。但在MySQL内部，为了

减少占用的空间以及方便计算，需要使用source_id的地方不是直接使用uuid的，而是使用一个int32的整数（sidno）来替代。既然使用整数来替代uuid，那么就需保存整数与uuid之间的相互映射关系，以便在命令SHOW MASTER STATUS或者SHOW SLAVE STAUTS的输出中显示uuid而不是看不懂的整数值，而类Sid_map就是用来保存这种映射关系的。MySQL内部使用一个64位的整数（gno）来作为GTID的transaction_id部分（第二部分）。类Sid_map的定义如下：

```
// sql/rpl_gtid.h

typedef Uuid rpl_sid;                                    // 使用uuid作为GTID的source_id
typedef int32 rpl_sidno;                                 // GTID的第一部分
typedef int64 rpl_gno;                                   // GTID的第二部分

struct Gtid
{
    rpl_sidno sidno;
    rpl_gno gno;
    ...                                                  //省略一些函数
};

class Sid_map
{
private:
    HASH _sid_to_sidno;                                  // rpl_sid到rpl_sidno的映射
    DYNAMIC_ARRAY _sidno_to_sid;                         // rpl_sidno到rpl_sid的映射
    mutable Checkable_rwlock *sid_lock;                  // 读写锁保护

public:
    rpl_sidno add_sid(const rpl_sid &sid);               // 添加sid
    rpl_sidno sid_to_sidno(const rpl_sid &sid) const;    // 获取rpl_sid到rpl_sidno的映射
    const rpl_sid &sidno_to_sid(rpl_sidno sidno) const;  // 获取rpl_sidno到rpl_sid的映射
    rpl_sidno get_max_sidno() const;                     // 获取当前最大的rpl_sidno
};
```

类Gtid代表一个GTID，主要的成员为sidno和gno，分别代表GTID的source_id部分和transaction_id部分。

类Sid_map存储了uuid字符串与整数sidno之间的相互映射关系，下面我们给出该类中各个成员的含义。

- _sid_to_sidno：采用哈希表存储uuid字符串到整数sidno的映射关系。
- _sidno_to_sid：这个数组存储的是整数sidno到uuid字符串的映射关系。由于sidno是从1开始自增的，所以将sidno等于n对应的uuid存储在数组下标为n-1的位置。当需要获取sidno为n对应的uuid时，直接取数组下标为n-1的元素就可以了，这样就实现了从rpl_sidno到uuid的映射。
- sid_lock：读写保护锁。

下面我们给出该类中各个成员函数的含义。

- ❑ add_sid：添加uuid字符串到Sid_map中。
- ❑ sid_to_sidno：根据uuid字符串查询对应的整数sidno。
- ❑ rpl_sid &sidno_to_sid：根据整数sidno查询对应的uuid字符串。
- ❑ get_max_sidno：获取最大的sidno。

- 类Gtid_set

该类是GTID集合在MySQL中的实现，下面我们给出该类的定义，其中省略了一些成员和成员函数，仅仅将比较核心的成员和成员函数列举出来：

```
// sql/rpl_gtid.h

class Gtid_set
{
public:
    struct Interval                              //代表一个区间
    {
        rpl_gno start;                           // start和end表示了前闭后开的区间[start,end)
        rpl_gno end;

        bool equals(const Interval &other) const // 判断区间是否相等
        { return start == other.start && end == other.end; }

        Interval *next; // 指向下一个区间
    };

private:
    struct Interval_chunk // 预分配和批量分配Interval对象
    {
        Interval_chunk *next;
        Interval intervals[1];
    };

private:
    Sid_map *sid_map;                            // sidno和sid的映射关系
    DYNAMIC_ARRAY intervals;                     // 存储所有GTID
    Interval *free_intervals;                    // 空闲的Interval对象，可以重复利用
    Interval_chunk *chunks;                      // 用于管理批量分配的 Interval对象
    mutable Checkable_rwlock *sid_lock;          // 读写保护锁

public:
    void clear();

    // 添加/删除GTID
    enum_return_status _add_gtid(rpl_sidno sidno, rpl_gno gno);
```

```
    enum_return_status _remove_gtid(rpl_sidno sidno, rpl_gno gno);

    // 添加/删除GTID集合
    enum_return_status add_gtid_set(const Gtid_set *other);
    enum_return_status remove_gtid_set(const Gtid_set *other);

    // 判断是否包含某个GTID
    bool contains_gtid(rpl_sidno sidno, rpl_gno gno) const;

    // 判断是否是某个GTID集合的子集
    bool is_subset(const Gtid_set *super) const;

    // 判断两个集合是否有交集
    bool is_intersection_nonempty(const Gtid_set *other) const;
    // 求两个集合的交集
    enum_return_status intersection(const Gtid_set *other, Gtid_set *result);

    static bool is_valid(const char *text);          // 判断字符串是否是一个合法的GTID集合表示

    char *to_string() const;                          // 以字符串的形式表示GTID集合

    Sid_map *get_sid_map() const { return sid_map; }
};
```

图8-6给出了Gtid_set类的大概结构。

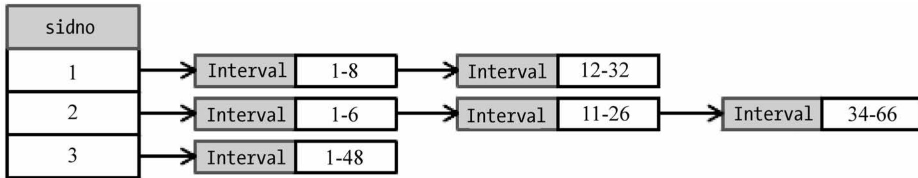


图8-6　Gtid_set类的结构示意图

结构体Interval代表一个前闭后开的整数区间，其成员start和end用于指明这个区间的范围，成员next用于实现Interval链表。

Interval_chunk结构主要用于预分配和批量分配Interval对象。

接下来，我们给出类Gtid_set中各个成员的含义以及各个成员函数的作用。

- sid_map：存储的是sidno与sid的相互映射关系。
- intervals：类Gtid_set的核心成员，是一个GTID集合。数组中的每个元素指向一个Interval链表，一个Interval链表代表了一个数据库实例上的所有GTID，所有的Interval链表组成了整个GTID集合，如图8-6所示。

- free_intervals：用于管理所有空闲的Interval对象。当需要获取Interval对象时，直接从free_intervals里获取；当删除Interval对象时，并不是直接删除而是加入free_intervals中，下次可以重复利用。
- chunks：结构体Interval_chunk的作用是预分配和批量分配Interval对象。当Gtid_set初始化时，会分配一个Interval_chunk对象，用于分配指定个数的Interval对象，然后将这些Interval对象全部放入free_intervals中，用于后续使用。例如，下面的语句是在初始化的时候预分配8个Interval对象（代码中8减去1的原因是结构体Interval_chunk的最后一个成员包含一个Interval对象）：

```
Interval_chunk *chunks = (Interval_chunk*) malloc (sizeof(Interval_chunk) + (8-1) * sizeof(Interval));
```

成员chunks管理所有已分配的Interval_chunk对象，在Gtid_set初始化的时候会通过分配一个Interval_chunk对象预分配n个Interval对象，然后将所有Interval对象加入到free_intervals中；当在使用Gtid_set的过程中发现free_intervals中已经没有空闲的Interval对象时，这时会分配一个新的Interval_chunk对象，将其中的所有Interval对象加入到free_intervals中；在Gtid_set销毁的时候，chunks中的所有Interval_chunk对象都会被释放掉。

下面我们看一下Gtid_set中各个成员函数的作用。

- _add_gtid：向Gtid_set中添加一个GTID。
- _remove_gtid：从Gtid_set中删除一个GTID。
- add_gtid_set：向Gtid_set中添加一个GTID集合。
- remove_gtid_set：从Gtid_set中删除一个GTID集合。
- contains_gtid：判断某个GTID是否在当前Gtid_set中。
- is_subset：判断当前Gtid_set是否是另一个Gtid_set的子集。
- is_intersection_nonempty：判断两个Gtid_set的交集是否不为空。
- intersection：求两个Gtid_set的交集。
- bool is_valid：判断某个字符串是否是一个GTID集合的合法表示。
- to_string：将当前Gtid_set表示成字符串形式。
- get_sid_map：获取包含的sid_map。

- 类Gtid_state

在MySQL中，不是直接使用Gtid_set类，而是使用Gtid_state类来管理各种GTID集合。Gtid_state类在全局只有一个实例Gtid_state *gtid_state。下面我们给出该类的定义：

```
// sql/rpl_gtid.h

class Gtid_state
{
private:
```

```
    Gtid_set logged_gtids;                  // 所有被执行并且记录到binlog的GTID集合
    Gtid_set lost_gtids;                    // 被删除的binlog中的GTID集合，lost_gtids永远
                                            // 是logged_gtids的子集
    Owned_gtids owned_gtids;                // 被线程持有的GTID集合

    mutable Checkable_rwlock *sid_lock;     // 读写保护锁
    mutable Sid_map *sid_map;

    rpl_sidno server_sidno;                 // 当前MySQL的sidno

public:
    void init();                            // 初始化

    // 判断指定的GTID是否属于logged_gtids
    bool is_logged(const Gtid &gtid) const;
    // 获取持有指定GTID的线程号
    my_thread_id get_owner(const Gtid &gtid) const;
    // 为线程thd获取指定的GTID
    enum_return_status acquire_ownership(THD *thd, const Gtid &gtid);

    // 自动生成GTID
    rpl_gno get_automatic_gno(rpl_sidno sidno) const;

    // 系统初始化时添加lost_gtids
    enum_return_status add_lost_gtids(const char *text);

    const Gtid_set *get_logged_gtids() const { return &logged_gtids; }
    const Gtid_set *get_lost_gtids() const { return &lost_gtids; }
    const Owned_gtids *get_owned_gtids() const { return &owned_gtids; }
    ...
};
```

接着我们给出类Gtid_state中各个成员的含义。

- logged_gtids：包含了该MySQL实例上执行过的并且记录在binlog中的所有GTID。
- lost_gtids：包含了已经删除的那些binlog文件中包含的所有GTID。lost_gtids永远是logged_gtids的子集。
- owned_gtids：管理所有正在被线程使用的GTID集合。

下面是类Gtid_state中各个成员函数的作用。

- is_logged：判断指定的GTID是否包含在logged_gtids内。
- get_owner：获取拥有指定GTID的线程。
- acquire_ownership：为线程thd获取指定的GTID。
- get_automatic_gno：该函数的作用是自动生成下一个GTID的gno，GTID的sidno由参数指定。该函数会检查logged_gtids的内容，生成一个最小的gno，保证该GTID在logged_gtids和owned_gtids中都没有出现。例如，当前logged_gtids中的内容是1:1-8:12-32,2:1-6:11-

26:34-66,3:1-48，我们想要通过该函数自动生成sidno为2的下一个GTID的gno。我们看到，sidno为2的Gtid_set里面包含了3个区间1-6、11-26以及34-66，我们会取6作为下一个GTID的候选gno（注意这里取的是6而不是7，这是因为该区间是前闭后开的）。如果GTID{2:6}没有包含在owned_gtids内，说明该GTID还没有被任何人使用，就返回该GTID；如果包含在owned_gtids中，说明该GTID正在被别人使用，这时我们取下一个GTID{2:7}，然后看该GTID是否被别的线程所使用，依次类推，直到找到一个没有使用的GTID。该函数会在事务提交到binlog时被调用，用来生成一个即将写入到binlog中的GTID事件。

- add_lost_gtids：系统初始化的过程中，会调用该函数将已删除binlog文件中包含的GTID集合添加到lost_gtids中。

2. GTID在MySQL中是如何工作的

介绍完MySQL实现中GTID相关的几个数据结构之后，接下来我们分别从系统启动到事务执行的过程中GTID机制在MySQL内部是如何运作的以及在复制过程中GTID是如何发挥作用的这几个方面介绍。

- *系统启动时*

在MySQL启动的过程中，如果配置了--gtid_mode=on参数，则会调用gtid_server_init函数分配全局变量Gtid_state *gtid_state。前面我们提到gtid_state是Gtid_state类在全局的唯一实例，后续所有对GTID的操作都是基于该实例的。

接下来会调用mysql_bin_log的init_gtid_sets函数，使用binlog的内容来初始化gtid_state的logged_gtids成员和lost_gtids成员。前面介绍过，在GTID开启的模式下，我们会在binlog中额外记录两种事件——Previous_gtids事件和Gtid事件。Previous_gtids事件记录在binlog的开头，用来记录在该binlog文件之前所有执行过的事务对应的GTID集合；在binlog中记录每个事务之前，都会先写入一个Gtid事件与该事务相对应。下面给出了init_gtid_sets函数的主要工作。

- 收集所有binlog文件名，按顺序放入一个链表。
- 读取链表中最新binlog文件中的Previous_gtids事件和所有的Gtid事件，并将这些事件代表的GTID集合加入到logged_gtids中。
- 读取链表中最老binlog文件的Previous_gtids事件，将事件表示的GTID集合加入到lost_gtids中。

如果配置了复制，则在系统启动阶段会调用init_slave函数来初始化复制相关的内容。如果同时开启了GTID，init_slave函数会读取relay-log的内容，将Previous_gtids事件以及Gtid事件记录的GTID信息记录下来，在从库请求主库发送binlog时会携带发送GTID信息，这样主库就知道哪些事务应该发送给从库，哪些不需要发送。当然，这只有在使用CHANGE MASTER TO命令时指定了MASTER_AUTO_POSITION=1的情况下才起作用。

● 事务提交时

从上面可以看到，当执行完init_gtid_sets函数之后，GTID相关的内容已经初始化完成了。下面我们看一下事务提交时GTID机制是如何工作的。

我们在第6章中介绍过，一个事务的所有修改产生的事件不会直接写入到binlog，而是暂时缓存在binlog_cache_mngr结构中，只有该事务提交时才会把binlog_cache_mngr中的内容完整地写入到binlog中。在GTID模式下，当事务提交时，在将binlog_cache_mngr内的内容写入到binlog之前，会插入一个Gtid事件到binlog_cache_mngr中与该事务对应，然后再将binlog_cache_mngr中的内容全部写入到binlog中。

● binlog发生切换时

当binlog发生切换时，通常情况下会在新的binlog文件的开头记录一个Format_desc事件，然后该binlog文件会成为当前活跃的binlog文件，所有后续的事务产生的事件都会被写入到该文件。在GTID模式下，当发生binlog切换时，在写入一个Format_desc事件后，紧接着会记录一个Previous_gtids事件，将logged_gtids中包含的GTID集合记录下来，代表当前已经执行过的所有GTID集合。

● binlog发生删除操作时

当binlog发生删除操作时，我们需要更新lost_gtids的内容，将所有已经删除的binlog文件包含的GTID添加到lost_gtids中去。

● 复制中GTID的作用

下面我们介绍一下在GTID模式下，MySQL中与复制相关的3个线程各自分别做了哪些额外的工作，进一步加深对基于GTID模式复制工作机制的理解。

(1) slave IO线程

在连接到主库之后，调用get_master_version_and_clock函数获取主库的版本和时钟信息时，也会查询主库是否支持GTID模式，并将结果记录下来以便后续使用。

接下来调用request_dump函数请求主库发送binlog事件。通常情况下，该函数会发送COM_BINLOG_DUMP命令，并将发送binlog的开始位置告诉主库。如果使用CHANGE MASTER TO命令时指定了MASTER_AUTO_POSITION = 1，将会发送COM_BINLOG_DUMP_GTID命令给主库，同时将当前logged_gtids和relay-log包含的GTID集合发送给主库，让主库知道从库已经接收了哪些事务。

从库获取主库的binlog事件后，会将其记录到relay-log中。如果获取的是Gtid事件，该事件也会被写入到relay-log中。

(2) slave SQL线程

在GTID模式下，我们必须开启log-slave-updates选项，这样当从库的slave SQL线程重放relay-log的内容时，会将对数据库的更新记录到binlog中，同时将原的Gtid事件（不是新生成的Gtid事件，因为该GTID对应的事务是在主库上提交的，从库只是重放而已）写入到binlog中。除此之外，还会将该GTID添加到logged_gtids中。所以，当我们在从库上使用SHOW BINLOG EVENTS命令查看binlog事件时，可能看到sidno不同的Gtids事件，它们分别对应于主库上提交的事务和从库上提交的事务。

(3) master dump线程

当主库收到COM_BINLOG_DUMP_GTID命令时，会执行com_binlog_dump_gtid函数，向从库发送binlog事件。该函数解析从库发送过来的GTID集合，根据该集合来定位发送binlog事件的开始位置。

下面我们来简单描述一下master dump线程是如何根据从库发送过来的GTID集合来自动判断从哪儿开始发送binlog事件。

① 调用mysql_bin_log的find_first_log_not_in_gtid_set函数，找到第一个含有从库未接收事务的binlog文件，说明从库之前的接收工作进行到了该文件的某个位置。

② 读取该binlog文件，如果遇到Gtid事件，查看该GTID是否包含在从库发送过来的GTID集合中，如果在，说明该事务已经发送给从库，则跳过该事务；如果没有在，则说明从库上次没有复制该事务，将其发送给从库。

8.9.4 MariaDB中的GTID

由于MariaDB支持多源复制，也就是说一个从库可能包含多个主库，为了在从库上区分多个主库，让复制工作能够正常进行，MariaDB中的GTID由三部分组成：域ID（domain id）、服务器ID（server id）和序列号（sequence number）。域ID的主要作用是区分多源复制中的多个主库，服务器ID和序列号的作用与MySQL中GTID的源ID和事务ID相同。

虽然在MariaDB中GTID的表示方法和MySQL有所不同，但整体的工作机制比较类似，这里我们就不做进一步介绍了，有兴趣的读者可以自行查阅官方文档，详见https://mariadb.com/kb/en/mariadb/mariadb-documentation/replication-cluster-multi-master/replication/global-transaction-id/。

8.10 小结

本章中，我们介绍了MariaDB/MySQL的复制功能，分析了复制的工作机制及其实现，并且详细讲解了半同步复制以及MariaDB的多源复制相关的内容。接下来，我们对比较重要的一些概

念进行简短的回顾。

- 复制有助于我们构造高性能的应用，也为高可用性、可扩展性、备份、灾难恢复等工作提供了可行的方案。
- 复制是基于二进制日志进行的，从库通过获取主库的binlog事件，然后在本地进行重放，使从库拥有与主库相同的数据。要想使用复制功能，必须开启binlog。
- 复制的工作主要由3个线程完成的：主库上的master dump线程负责将binlog的更新发送到从库上；从库的slave IO线程负责接收主库发送过来的binlog事件，并且将其写入到relay-log中；从库的slave SQL线程负责对relay-log中的事件进行重放。
- master.info文件存储的是从库连接主库需要的所有信息以及从库接收主库binlog事件的进度等信息。relay_log.info文件记录的是从库重放的进度等信息。正是有了这些信息，我们才能很容易地使用`STOP SLAVE`命令停止复制，进行相应的分析或者备份等工作，然后使用`START SLAVE`命令重新开启复制。
- 从MariaDB/MySQL 5.5开始，MariaDB/MySQL以插件的形式引入了半同步复制功能。当至少一个从库收到更新之后，主库才将提交的事务返回给客户端。半同步复制很好地解决了主从数据一致性的问题。
- MariaDB 10.0引入了多源复制的功能，允许一个从库可以有多个主库，该功能提供了将多个实例的数据聚集到一块进行分析处理或者使用一个实例对多个实例的数据进行备份的方便途径。为了实现多源复制，MariaDB对一些命令做了扩展，同时新增了几个命令，例如`START ALL SLAVES`、`STOP ALL SLAVES`等。
- MySQL 5.6.5引入了GTID特性，使复制的相关工作变得更加简单。MariaDB在10.0.2中也引入了GTID特性。

第 9 章

数据结构和算法

9

数据结构和算法是程序的灵魂，前者表示如何组织你的数据，而后者表示如何处理这些数据。合适的数据结构和算法能够让你的程序性能提高数十倍、数百倍，甚至更多。

本章中，我们将对数据库中使用到的重要数据结构以及MariaDB/MySQL的一些复杂查询`ORDER BY`、`JOIN`等使用到的算法进行分析。

本章的内容主要包括：

- 算法复杂度
- B+树和索引
- 堆与排序算法
- `order by`的实现
- `join`的实现

9.1 算法复杂度

算法是定义良好的计算过程，它取一个或一组值作为输入，并产生一个或一组值作为输出。也就是说，算法是一系列计算步骤，用来将输入数据转换成输出结果。

算法的复杂度分为时间复杂度和空间复杂度，它们是衡量一个算法好坏的两个不同维度的指标。

1. 时间复杂度

算法的时间复杂度描述的是算法运行时占用的时间随问题规模的变化而变化的规律。一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，我们记为$T(n)$，若某个辅助函数$f(n)$使得当n趋近于无穷大时，$T(n)/f(n)$的极限值为不等于0的常数，则称函数$f(n)$是$T(n)$的同数量级函数，记作$T(n)=O(f(n))$，而$O(f(n))$为算法的渐进时间复杂度，简称为时间复杂度。

常见的时间复杂度有：常数复杂度$O(1)$、对数复杂度$O(\lg n)$、线性复杂度$O(n)$、线性对数复

杂度$O(n\lg n)$、平方复杂度$O(n^2)$、立方复杂度$O(n^3)$、指数复杂度$O(2^n)$，等等。如果算法中执行语句的次数是固定的，不会随问题的规模增长而增长，则我们称该算法具有常数时间复杂度，记为$O(1)$，例如哈希表的查找和插入操作。

2. 空间复杂度

空间复杂度是对算法在执行过程中占用的临时存储空间大小的度量，它也是问题规模n的函数。一个算法所占用的存储空间主要包括输入输出数据所占的空间和算法运行过程中临时占用的存储空间。输入输出占用的存储空间由问题的规模决定，不会随算法的不同而改变，而算法在运行过程中临时占用的存储空间根据算法的不同而各不相同，有的算法只需占用少量临时存储空间，不随问题规模的增大而增加；有的算法占用的临时空间随问题规模的增大而增加，当n较大时将占用较多的存储空间。

在很多情况下，一个算法的时间复杂度和空间复杂度往往是相互矛盾的，时间复杂度小的算法经常具有较大的空间复杂度，而时间复杂度大的算法经常具有较小的空间复杂度。我们不能一味地说某个算法是好还是坏，需要根据具体的应用场景来定，只要能够满足我们需求的都是好算法。例如，如果我们的存储空间是非常宝贵和有限的资源，但对算法的执行速度要求不是那么严格的情况下，应当选择空间复杂度较小的算法，即使该算法具有较高的时间复杂度，只要在我们能够接受的范围内就可以，因为这种情况下存储空间是比时间更珍贵的资源。如果我们对算法的执行速度有非常严格的要求，而存储空间非常充裕，那么应该选择时间复杂度小的算法，即使它会占用更多的临时空间，因为在这种情况下时间是比空间更珍贵的资源。算法的选取其实就是权衡什么才是我们更珍贵的资源过程，要么以时间资源换取空间资源，要么以空间资源换取时间资源。

9.2　B+树和索引

数据结构是存储和组织数据的一种方式，以便于对数据进行访问和修改。没有哪种数据结构可以适用于所有的用途和目的，每种数据结构都有自己的长处和局限性。本节中，我们将介绍一种为磁盘或其他直接存取辅助存储设备而设计的一种平衡查找树——B+树。B+树能够降低磁盘的I/O次数，在数据库以及文件系统中得到了广泛使用。

9.2.1　磁盘的读取

图9-1显示的是一个典型的磁盘驱动器，这个驱动器主要包括若干盘片，它们以固定的速度绕共用的主轴旋转。每个盘的表面覆盖一层可磁化的物质，每个盘片通过磁臂末端的磁头来读写数据。磁臂是通过物理手段连接在一起的，它们可以将磁头移近或者远离主轴。

虽然磁盘比主存便宜而且有较高的容量，但它们的运行速度很慢，因为它们有机械移动的部

分，包括磁臂移动和盘旋转。磁臂移动就是我们通常所说的寻道，在我们读取或者写入数据的时候，首先需要将磁头移动到正确的磁道上。其次就是盘旋转，目前主流的商用磁盘的旋转速度在7200 ~ 15 000转/分钟之间，旋转一圈需要花费4 ~ 8.3毫秒。

为了平摊等待机械移动所花费的时间，磁盘的一次读取操作通常会读取多个数据项，例如一次读取512字节、1024字节，等等。

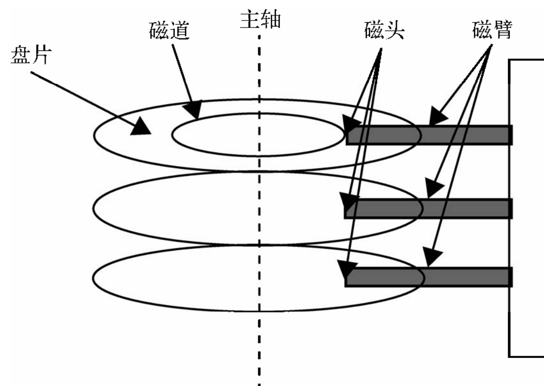


图9-1 磁盘驱动器

9.2.2 B+树

我们知道内存的一次存取操作需要几十纳秒到100纳秒，而一次磁盘I/O所需的时间大概在几毫秒到10毫秒，两者之间相差5个数量级。磁盘I/O操作是个耗时的过程，为了提高系统的整体性能，应该尽量减少系统的磁盘I/O次数。

B+树是为磁盘或其他直接存取辅助存储设备而设计的一种平衡查找树，它与红黑树和其他平衡二叉树类似，但在降低磁盘的I/O次数方面做得更好。B+树在对磁盘I/O非常敏感的系统中得到了非常广泛的应用，例如文件系统和数据库系统。

一棵B+树通常由一个根结点、多个非叶子结点和多个叶子结点组成。

B+树和以红黑树为代表的其他二叉树不同的地方在于，B+树的每个结点可以有许多子结点，从几个到几千个不等。也就是说，B+树的“分支因子”很大，这就决定了B+树的高度比含有相同结点数的红黑树的高度要小得多，减少了读取某个元素时需要进行的磁盘I/O次数。

B+树是B树的变种，与B树相比，B+树的一个最大不同点是所有的数据都存储在叶子节点中，非叶子节点只存储相应的键。B+树具有如下几个特点。

- 所有的关键字都出现在叶子结点中。
- 不可能在非叶子结点命中。

- 非叶子结点相当于叶子结点的索引，叶子结点相当于存储数据（关键字）的数据层。
- 数据在叶子结点这一层是按顺序排列的。

B+树从诞生以来得到了广泛的应用。在文件系统领域，NTFS、ReiserFS、NSS、XFS、JFS和ReFS等文件系统都使用B+树来索引文件。在关系型数据库领域，IBM DB2、Microsoft SQL Server、Oracle、SQLite以及MySQL/MariaDB中非常多的存储引擎都使用B+树来作为数据的索引，提高数据的查询速度。著名的键/值存储引擎Tokyo Cabinet也支持B+树类型的索引。

图9-2给出了一个高度为2的B+树，从图中可以看出，叶子结点不仅存储了关键字，而且还存储了关键字对应的数据。

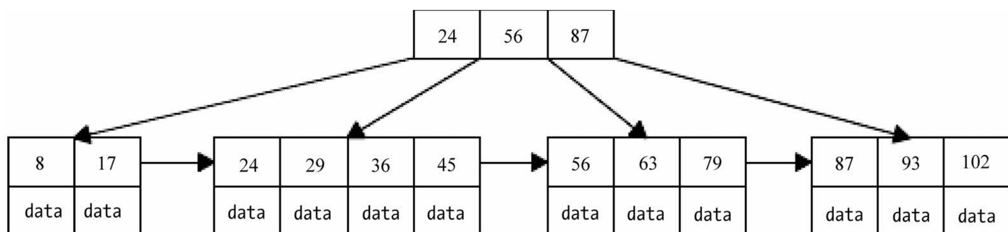


图9-2　B+树

9.2.3　数据库索引

数据库索引是帮助数据库高效获取数据的数据结构，也就是说，数据库索引的本质是一种数据结构，它能够提高查找数据的速度。

数据库索引能够帮助我们快速定位到所需的数据，而不需要遍历表的所有行。同时，索引增加了写操作的开销以及占用的存储空间，因为我们除了要写入相应的行数据，还需要修改对应的索引。当表的索引过多时，一方面会占用更多的磁盘空间，另一方面大大增加了写入的开销，所以通常建议一个表的索引个数不宜过多，不必要的索引应该删除。

索引分为聚簇索引和非聚簇索引两种，聚簇索引中表数据的物理存储顺序和索引的顺序一致，而非聚簇索引中表数据的物理存储顺序和索引的顺序无关。由于表数据的物理存储顺序只可能有一种，所以一个表只能有一个聚簇索引，但是可以有多个非聚簇索引。例如，一个InnoDB存储引擎的表可以有一个主键索引（聚簇索引）和多个二级索引（非聚簇索引）。

图9-3中给出了一个B+树索引的例子，根结点位于内存，其他的结点位于磁盘。在实际情况中，由于系统的内存是有限的，内存中只存储了索引的部分结点，而其他的结点则位于磁盘中，当查询的数据所处的结点在磁盘中时，我们需要将该结点从磁盘读入到内存中。

当我们查找关键字为60的数据时，首先从根结点开始查找，然后到结点A中查找，最后到结

点B中查找。由于结点A和B位于磁盘中，所以需要将它们读取到内存，这样就涉及两次磁盘IO操作。

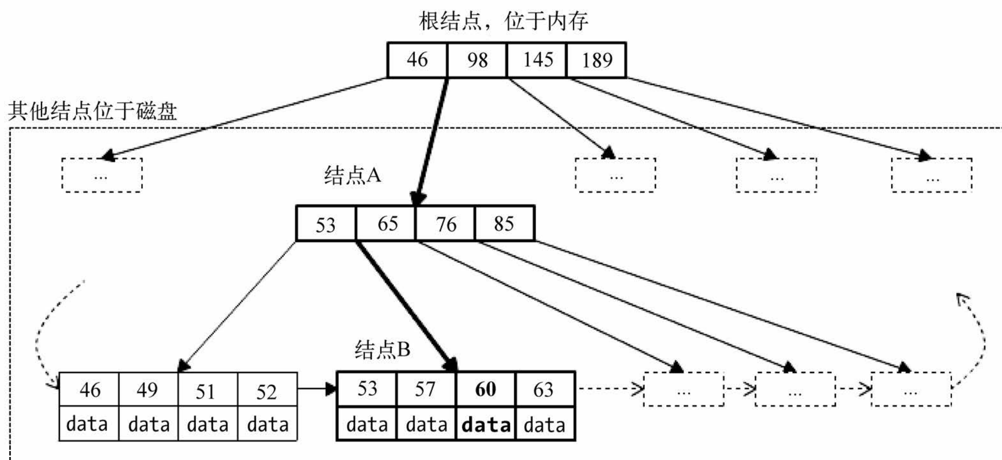


图9-3 B+树索引

B+树是一种顺序查找树，其中的数据是按顺序排列的。B+树类型的索引具有以下特点。

- **可以进行范围查找**。由于B+树中的数据是有序的，所以B+树类型的索引可以满足类似`where time between '2014-05-01' and '2014-07-01'`或者`where id < 1000`这样的范围查询。
- **支持全值匹配查询**。B+树类型的索引支持`where name = "Jim"`这样的全值匹配查询。
- **支持最左前缀匹配**。B+树类型的复合索引满足最左前缀匹配原则。

除了B+树索引，很多数据库引擎还支持哈希索引。哈希索引中的数据是无序的，只能用于精确匹配。

9.3 堆排序与快速排序

本节中，我们将介绍堆数据结构、堆排序算法和快速排序算法，为后面讲解`order by`的实现奠定基础。

9.3.1 堆——优先级队列

堆数据结构是一种数组对象，它可以被视为一颗完整的二叉树，树中的每个结点与数组中存放该结点值的那个元素对应。树的每一层都是填满的，除了最后一层。下面我们给出了堆数据结构中父结点以及左右孩子结点的下标计算公式。

父结点的下标计算公式：`parent(i) = i / 2`

左孩子结点下标计算公式：`left(i) = 2 * i`

右孩子结点下标计算公式：`right(i) = 2 * i + 1`

堆分为大根堆和小根堆两种：大根堆的最大元素在堆顶，父结点元素的值总是大于等于子结点的值；小根堆的最小元素在堆顶，父结点元素的值总是小于等于子结点元素的值。图9-4给出的是一个大根堆的例子。

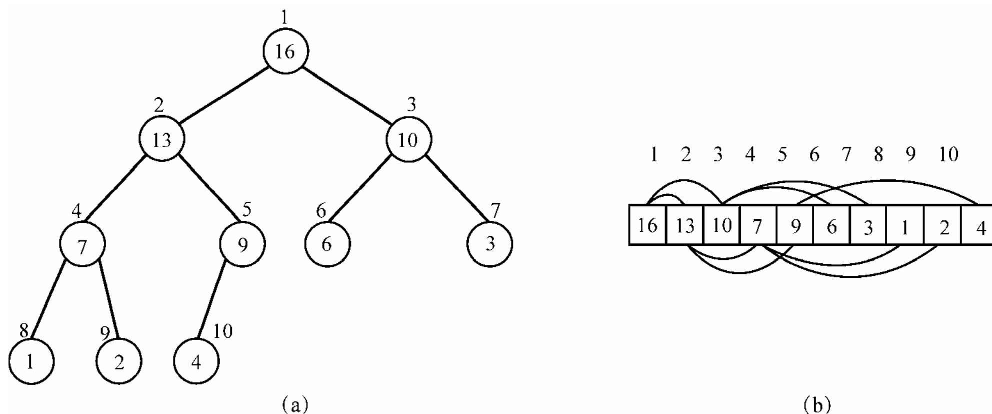


图9-4　堆数据结构（大根堆）。堆可以看作一棵二叉树(a)和一个数组(b)。圆圈中的值表示树中每个结点存储的值，结点上方的数字表示对应的数组下标。数组上方的连线表示父子关系

若不熟悉堆的创建、往堆里添加元素以及以堆顶取出最大（小）元素等操作，读者可以自行翻阅相关资料，这里不再过多介绍。

9.3.2　堆排序

堆排序算法由J. W. J. Williams在1964年发明的，它是一种利用堆（优先队列）这一数据结构进行排序的比较排序算法，其时间复杂度为$O(n\lg n)$。堆排序是一种能够在原地进行排序的排序算法。

堆排序的过程分为两个部分：建堆过程和排序过程。下面我们给出了堆排序的主要执行流程。

(1) 建立大根堆。

(2) i = 堆大小，交换堆顶元素与下标为i的元素的位置，堆大小减1，让堆顶元素在新堆（大小减1之后的堆）中下降，使新堆满足堆的性质。

(3) 重复执行步骤(2)直到堆的大小为1。所有步骤结束之后，数组中的元素将按升序排列，排

序过程结束。

如果想将元素按降序排列，在第(1)步中建立小根堆，其他步骤不变。

9.3.3 快速排序——qsort

快速排序是一种被广泛使用的比较排序算法，其平均时间复杂度为$O(n\lg n)$，最坏时间复杂度为$O(n^2)$，空间复杂度是$O(1)$。在实践中，快速排序算法通常比其他时间复杂度为$O(n\lg n)$的比较排序算法拥有更快的速度，因为快速排序算法具有更小的常数因子，所以快速排序是我们通常首选的比较排序算法。

快速排序算法由Tony Hoare于1960年发明的，当时Tony正在开发一个机器翻译的项目，项目中需要对待翻译的单词进行排序。快速排序算法是一种分治算法，首先将数组划分成两个子分区，左边子分区的所有元素小于右边子分区的所有元素，然后在子分区中递归这个划分过程，直到分区中只有一个元素。该算法的执行步骤如下。

(1) 从数组中选取一个数，作为基准数。

(2) 将小于基准数的所有数移动到基准数的左边，将大于等于基准数的所有数移动到基准数的右边。经过分区之后，原来的一个分区被划分成两个子分区，左分区的任何一个元素都小于右分区的任何一个元素。

(3) 在左右两个分区中递归执行步骤(1)和步骤(2)，直到分区只有一个元素。

下面我们给出快速排序的伪代码，其中:=表示赋值操作：

```
1. quicksort(A, i, k):
2.     if i < k:
3.         p := partition(A, i, k)
4.         quicksort(A, i, p-1)
5.         quicksort(A, p+1, k)

6. partition(array, left, right)
7.     pivotIndex := choose-pivot(arrary, left, right)
8.     pivotValue := array[pivotIndex]
9.     swap array[pivotIndex] and array[right]
10.    storeIndex := left
11.    for i from left to right-1
12.        if array[i] <= pivotValue
13.            swap array[i] and array[storeIndex]
14.            storeIndex := storeIndex + 1
15.    swap array[storeIndex] and array[right]
16.    return storeIndex
```

quicksort的过程相对比较简单：传入参数为数组A[i, k]，如果i < k，执行partition过程将数组分成两个子分区A[i, p-1]和A[p+1, k]，A[i, p-1]分区中所有元素的值小于等于A[p]的值，

A[p+1, k]分区中所有元素的值大于A[p]的值。然后在两个子分区上递归执行quicksort过程。

quicksort的核心是划分过程partition，接下来我们分析下partition的执行过程：伪代码第7行到第9行用于从数组中选取一个元素作为分区的基准数，并将该元素和数组的最后一个元素进行交换。伪代码第11行到第14行中for循环用于判断数组中元素的值与基准数的大小关系，如果元素的值小于基准数的值，则将元素的值交换到左边分区。最后将基准数放到数组中的合适位置，数组被该基准数分成了左右两个子分区。

图9-5给出了一个partition的例子。

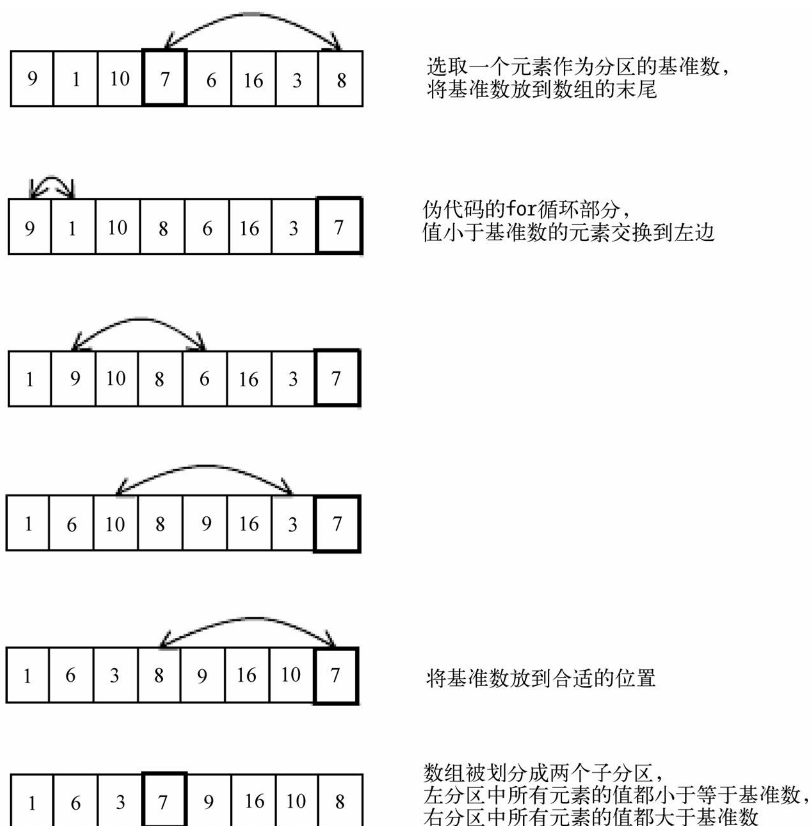


图9-5　快速排序的partition过程

9.4　ORDER BY 的实现

ORDER BY是MariaDB/MySQL使用比较频繁的SQL语句，用于对查询结果按照指定的字段进行排序。ASC和DESC关键字指明了结果集是按升序还是降序进行排列。ORDER BY偶尔也会和LIMIT关键

字组合使用，LIMIT关键字用于限制结果集的数量或者对结果集进行分页。ORDER BY的语法如下：

```
SELECT ... FROM tb [WHERE ...] ORDER BY ... [ASC | DESC] [LIMIT n,m]
```

下面给出的例子使用ORDER BY按照STATE字段对结果集进行排列：

```
mysql> SELECT * FROM userinfo ORDER BY state;
+-------+---------------------------+-------------------+
| id    | name                      | state             |
+-------+---------------------------+-------------------+
| 1     | jim                       | California        |
| 5     | J.Rod                     | California        |
| 2     | can                       | Florida           |
| 4     | billy                     | Florida           |
| 3     | Jordan                    | Kansas            |
| 6     | J.W.J Williams            | Kansas            |
+-------+---------------------------+-------------------+
6 rows in set (0.01 sec)
```

9.4.1 使用索引的已有顺序

通常情况下，当收到ORDER BY请求时，MariaDB/MySQL需要对结果集按照指定的字段进行排序，然后返回给客户端。当满足某种条件时，MariaDB/MySQL会利用数据库索引（这里指的是B+树类型的索引，因为只有B+树类型的索引才有顺序，所以本节中我们都假设索引类型为B+树类型）具有一定的顺序这一特性，按照索引顺序将结果返回给客户端，这样就省略了对结果集的排序过程，加快了查询的响应速度。

下面我们通过一个例子来讲解使用现有索引的已有顺序来满足ORDER BY操作必须具备的几个条件。我们创建了一个userinfo表，该表包含两个索引，一个主键索引（uid）和一个二级索引（create_time, name）：

```
CREATE TABLE `userinfo` (
    `uid` int(11) NOT NULL,
    `name` varchar(64) NOT NULL,
    `adress` varchar(128) DEFAULT NULL,
    `create_time` datetime DEFAULT NULL,
    `email` varchar(128) DEFAULT NULL,
    PRIMARY KEY (`uid`),
    KEY `idx_uni` (`create_time`,`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

下面我们给出几种ORDER BY语句利用索引的已有顺序进行排序的情况。

首先，使用WHERE语句和ORDER BY语句满足组合索引的最左前缀，并且查询的所有列都包含在该索引中，下面我们给出两条SELECT语句来验证这一点：

```
mysql> EXPLAIN SELECT * FROM userinfo WHERE create_time="2014-05-02" ORDER BY name;
+-------+------------------+-------------+---------+--------------------------------------+
| id    | select_type      | table       | ...     | Extra                                |
+-------+------------------+-------------+---------+--------------------------------------+
| 1     | SIMPLE           | userinfo    | ...     | Using where; Using filesort          |
+-------+------------------+-------------+---------+--------------------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT name FROM userinfo WHERE create_time="2014-05-02" ORDER BY name;
+-------+------------------+-------------+---------+--------------------------------------+
| id    | select_type      | table       | ...     | Extra                                |
+-------+------------------+-------------+---------+--------------------------------------+
| 1     | SIMPLE           | userinfo    | ...     | Using where; Using index             |
+-------+------------------+-------------+---------+--------------------------------------+
1 row in set (0.00 sec)
```

这里我们主要关注EXPLAIN输出的Extra列的结果，Using index表示使用索引的已有顺序来满足ORDER BY请求，Using filesort表示使用filesort算法来进行实实在在的排序操作，对结果集进行排序来满足ORDER BY请求。

第一条SELECT语句中，虽然WHERE语句和ORDER BY语句的组合满足组合索引的最左前缀，但是查询的列没有全部包含在组合索引中。而第二条SELECT语句中，查询的列都包含在索引中，所以使用了索引的已有顺序来对结果集进行排序。

其次，想要利用索引的已有顺序来满足ORDER BY请求，order by各个字段必须包含在同一个索引中，并且字段的顺序必须和索引的顺序完全一致或完全相反。下面我们给出例子来证明这一点：

```
mysql> EXPLAIN SELECT name FROM userinfo ORDER BY create_time, name;
+--------+------------------+------------------+----------+----------------------------------+
| id     | select_type      | table            | ...      | Extra                            |
+--------+------------------+------------------+----------+----------------------------------+
| 1      | SIMPLE           | userinfo         | ...      | Using index                      |
+--------+------------------+------------------+----------+----------------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT name FROM userinfo ORDER BY create_time desc, name desc;
+--------+------------------+------------------+----------+----------------------------------+
| id     | select_type      | table            | ...      | Extra                            |
+--------+------------------+------------------+----------+----------------------------------+
| 1      | SIMPLE           | userinfo         | ...      | Using index                      |
+--------+------------------+------------------+----------+----------------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT name FROM userinfo ORDER BY create_time ASC, name DESC;
+--------+------------------+------------------+----------+----------------------------------+
| id     | select_type      | table            | ...      | Extra                            |
```

```
+--------+------------------+------------------+----------+----------------------------------+
| 1      | SIMPLE           | userinfo         | ...      | Using index; Using filesort      |
+--------+------------------+------------------+----------+----------------------------------+
1 row in set (0.00 sec)
```

第一条SELECT语句按照create_time和name进行升序排列，第二条SELECT语句按照create_time和name进行降序排列，都可以利用索引的已有顺序来满足ORDER BY请求，而第三条SELECT语句混合使用ASC和DESC来修饰ORDER BY的字段，导致不能使用索引的已有顺序来满足ORDER BY的请求。

9.4.2 filesort算法

当不能使用已有索引的顺序来满足ORDER BY请求时，MariaDB/MySQL使用filesort算法对查询结果进行实实在在的排序操作，下面简要介绍一下这个算法。

1. filesort算法的执行流程

图9-6给出了filesort算法的执行流程。当结果集比较小时，排序缓冲区能够存储所有匹配查询的行，直接在排序缓冲区中进行快速排序，然后返回给客户端。这种情况下，排序动作发生在内存中，效率会比较高。当结果集比较大的时候，排序缓存区不能存储所有匹配查询的行，读取部分匹配的行到排序缓冲区，然后进行快速排序，将排好序的结果集写入到临时文件中，重复这个过程，直到所有匹配的行都写入到了临时文件，最后对临时文件中的数据进行归并排序，得到最终的结果。

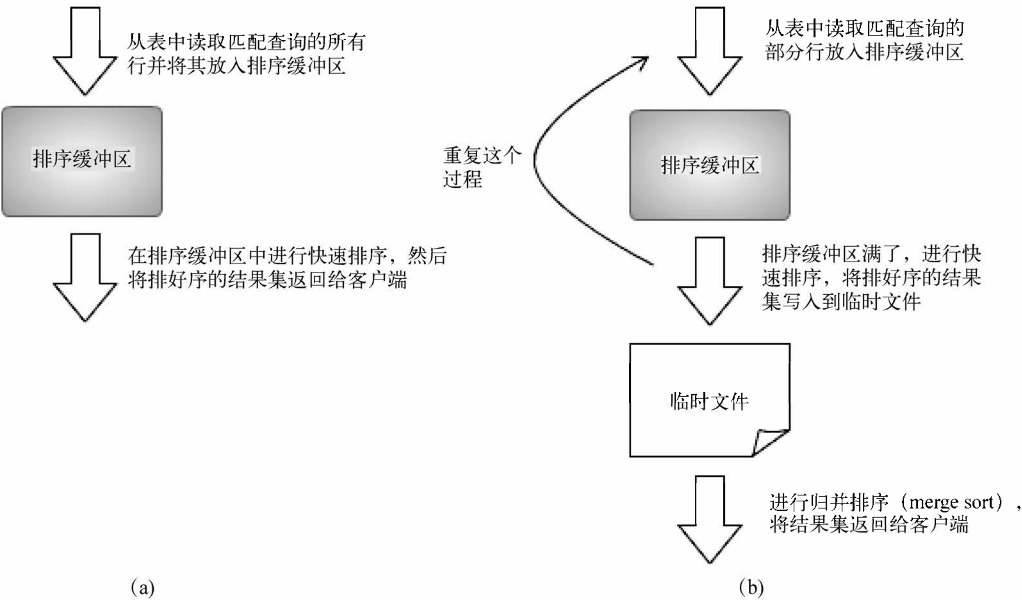


图9-6 filesort算法

9

2. filesort相关的参数

下面我们介绍MariaDB/MySQL中两个和filesort排序算法相关的参数sort_buffer_size和max_length_for_sort_data。

- sort_buffer_size参数。该参数的作用是设置filesort算法排序缓冲区的大小，它直接决定了排序操作的效率。当排序缓冲区不足以存储所有匹配查询的行时，filesort算法需要借助临时文件来存储中间结果集，这会导致对磁盘文件的I/O操作，从而降低排序操作的效率。从这方面来考虑，sort_buffer_size参数的值越大越好，因为这样可以保证所有的排序动作都发生在内存中。但是，从另一方面来讲，系统资源是有限的。同时，有个需要注意的地方是，MariaDB/MySQL每个连接有自己独立的排序缓冲区。sort_buffer_size参数指定的是每个单独排序缓冲区的大小，没有参数限制所有排序缓冲区占用的内存大小，所以当连接数比较多，并且其中有大量的连接正在进行ORDER BY操作时，将会导致排序缓冲区占用大量的内存，导致系统资源紧张。所以应该根据自己的具体应用场景，为sort_buffer_size选择合适的值。
- max_length_for_sort_data参数。当在内存中排序时，filesort算法会根据查询语句所取出的所有字段的长度总和与参数max_length_for_sort_data值的大小关系来选择两种不同的策略进行排序。如果前者大，filesort算法首先取出排序字段以及可以定位行数据的行指针信息，然后在排序缓冲区中进行排序，最后根据行指针信息取出其他字段的值。这种方式的排序优点是行数据进行了压缩（仅仅取出了排序相关的字段以及行指针），同样大小的排序缓冲区可以存储更多的行；缺点是需要两次访问表数据，第一次从表中取出排序字段和行指针，第二次是根据排好序的行指针，从表中取出其他字段的值。当max_length_for_sort_data的值大于查询语句中所查询的所有字段的长度总和时，filesort算法会一次性地从表中取出所查询的所有字段，然后在排序缓冲区中进行排序操作。这种方式的优点是只访问了一次表数据，减少了访问表数据导致的I/O操作；缺点是由于行数据没有进行压缩，会耗用更多排序缓冲区的空间。

3. filesort算法对包含LIMIT关键字的优化

当ORDER BY查询中包含LIMIT关键字时，如果LIMIT关键字后面的参数n与m的和比匹配查询的结果集小很多时，filesort算法内部会采取堆排序算法而不是快速排序算法在内存中对行数据进行排序：

```
SELECT * FROM tb1 [WHERE ...] ORDER BY col LIMIT n, m
```

在具体介绍MariaDB/MySQL在对包含LIMIT关键字的ORDER BY做的优化之前，我们先来看一个问题：如何在10亿个整数中获取最大的100个整数？

下面我们给出几种解决该问题的方案。

● 方案1

第一个是容易想到的解决方案如下，我们将其称为方案1。

(1) 对10亿个整数进行降序排序。
(2) 取出前100个整数。

由于整数的范围比较大，不能使用像计数排序这样拥有线性时间复杂度的排序算法，只能采用比较排序算法进行排序。步骤(1)的时间复杂度为$O(n\lg n)$（当然如果10亿个整数不能存储在内存中，还需要借助磁盘文件来存储中间结果，然后进行归并排序，那样的话花费的时间更多），步骤(2)的时间复杂度为$O(1)$，所以方案1的时间复杂度为$O(n\lg n)$。

该方案对应于`SELECT * FROM ... ORDER BY col LIMIT n, m`中的`n+m = 100`，而匹配查询的行数为10亿行的情况下，使用快速排序算法对所有数据进行排序，然后取出前面的100条数据。

● 方案2

假设有一种方法能够从一组数中找到任意第k大的数，那么我们又可以想到另一种方案来解决上面提出的问题，我们将其称为方案2。该方案的执行流程如下：

找出最大的值，找出第2大的值，找出第3大的值，……找出第100大的值。

确实存在这样的方法，可以找出数组中第k大的数，我们称该过程为`select`。`select`利用了快速排序中的`partition`过程。下面我们给出`select`的伪码：

```
1. select(arry, left, right, m)
2.     if left == right
3.         return array[left]
4.     p := partition(array, left, right)
5.     pivotOffset = p - left + 1
6.     if m == pivotOffset
7.         return array[p]
8.     elseif m < pivotOffset
9.         return select(array, left, p-1, m)
10.    else
11.        return select(array, p+1, right, m-pivotOffset)
```

假设问题的规模为n，那么`select`过程需要进行约$2n$次比较操作，也就是说`select`的时间复杂度为$O(n)$。那么方案2只需要执行100次`select`过程，就能找出前100个整数，进行的比较次数大约为$200n$，那么方案2的时间复杂度为$O(n)$。虽然方案2具有线性时间复杂度，但存在如下两个问题。

- n前面的系数比较大（200）。
- 需要遍历所有元素100次，如果n的规模比较大，内存无法存储所有的元素，就必须使用磁盘文件来存储这些元素，此时每次遍历所有元素就会涉及磁盘文件的I/O操作。

- 方案3

是否还存在一个更好的方案呢？既能有线性的时间复杂度，并且只需要遍历所有元素一次。答案是肯定的，这就是我们接下来要介绍的，那就是利用堆数据结构，我们称该方案为方案3。

方案3的执行步骤如下。

(1) 创建一个最多可以容纳100个元素的小根堆（优先队列）。

(2) 遍历所有的整数，如果堆未满，将元素加入到堆中；如果堆已满，比较该整数与堆顶整数的大小关系，如果小于堆顶的整数，丢弃该整数，如果大于堆顶的整数，以该整数替换掉堆顶的整数，让其在堆中下降。

方案3需要进行n次比较操作，最坏的情况出现在n个整数是按照升序进行排列的，每次比较完之后都需要替换掉小根堆堆顶的元素，然后该元素在堆中进行下降。结点数为100的堆对应的树的深度为7，那么每次下降过程最多下降6层，也就是说方案3在最坏情况下需要进行n次比较和$6n$次下降操作。所以方案3的时间复杂度为$O(n)$，系数在1至7之间，只需要遍历一次所有的元素。

经过以上的分析我们知道，三种方案中方案3是最快的方案，它对应于`SELECT * FROM ... ORDER BY col LIMIT n, m`中的`n+m = 100`而匹配的行数为10亿行时，采用一个小根堆来存储最大的100个结果，然后采用堆排序将堆中的100行记录进行排序。这也就是我们接下来要介绍的MariaDB/MySQL针对`LIMIT`所做的优化。

MariaDB/MySQL在执行filesort算法对结果集进行排序的过程中，会调用`check_if_pq_applicable`函数来判断是否需要采用堆排序来替代快速排序。该函数主要做了以下几个判断。

(1) 当排序缓冲区能够容纳的记录数大于预计匹配查询的行数时，如果`limit`中的`n+m < C / 3`，使用堆排序，否则使用快速排序，其中C表示预计匹配查询的行数。通过一系列的测试，在内存中对同样的数据集进行排序，快速排序的速度是堆排序的3倍左右，所以公式中采用系数3来衡量是使用快速排序算法还是堆排序算法。

(2) 当排序缓冲区能够容纳的记录数大于`n+m`但是小于预计匹配查询的行数时，MariaDB/MySQL会评估是采用堆排序快还是采用快速排序加归并排序快，然后选择合适的方案。评估终归是评估，结果不可能百分百正确，可能在一些情况下会出现错误的评估，这没有关系，只要大多数情况下是正确的，就能够提高系统的整体性能。

(3) 当排序缓冲区不能容纳`n+m`行数据时，使用快速排序。

9.5　JOIN 的实现

为了得到完整的结果，我们需要从两个或者更多表中获取数据，这时我们就需要执行`JOIN`语句对两个或者多个表进行连接操作。本节中，我们将介绍MariaDB/MySQL中`JOIN`语句及其用到的相关算法。

9.5.1 JOIN语句的使用

JOIN语句用于对两个或更多的表进行连接操作，它可以携带WHERE条件，用于对结果进行过滤。JOIN的语法如下，ON后面的短语我们称为连接条件（join condition）：

```
SELECT * FROM tb1[INNER | LEFT | RIGHT | CROSS] JOIN tb2 [ON tb1.a=tb2.b] [WHERE ...]
```

JOIN操作包括INNER JOIN、LEFT JOIN、RIGHT JOIN和CROSS JOIN四种类型。下面我们给出两张表table_a和table_b的记录：

```
table_a
id    name
1     steven
2     king
3     lily
5     suarez
6     halo

table_b
id    time
2     20140702
3     20140629
4     20140520
5     20140722
```

1. INNER JOIN

INNER JOIN也叫内连接、相等连接或者等值连接，它将符合连接条件的结果显示出来。在MariaDB/MySQL中，JOIN和INNER JOIN是等价的。下面给出了对上面两张表进行内连接的结果：

```
SELECT * FROM table_a INNER JOIN table_b ON table_a.id = table_b.id

id    name      id    time
2     king      2     20140702
3     lily      3     20140629
5     suarez    5     20140722

SELECT * FROM table_a AS a INNER JOIN table_b AS b ON a.id = b.id WHERE b.time > "20140701"

id    name      id    time
2     king      2     20140702
5     suarez    5     20140722
```

2. LEFT JOIN

LEFT JOIN也叫左连接，它以左边表格为基准进行连接，左表的所有记录都会显示出来，而

右表只会显示符合连接条件的记录，右表不足的记录都会以NULL填充，示例如下：

```
SELECT * FROM table_a AS a LEFT JOIN table_b AS b ON a.id = b.id

id     name       id     time
1      steven     NULL   NULL
2      king       2      20140702
3      lily       3      20140629
5      suarez     5      20140722
6      halo       NULL   NULL
```

3. RIGHT JOIN

RIGHT JOIN也叫右连接，它和左连接相反，以右表作为基准进行连接，右表的所有记录都会显示出来，而左表只会显示符合连接条件的记录，左表不足的记录以NULL进行填充，示例如下：

```
SELECT * FROM table_a AS a LEFT JOIN table_b AS b ON a.id = b.id

id     name       id   time
2      king       2    20140702
3      lily       3    20140629
NULL   NULL       4    20140520
5      suarez     5    20140722
```

4. CROSS JOIN

CROSS JOIN也叫交叉连接，在SQL标准中它是对两个表的记录做笛卡尔积，但在MariaDB/MySQL中，CROSS JOIN和INNER JOIN是等同的。

9.5.2　Nest Loop Join算法

正如名字一样，Nest Loop Join算法就像我们写程序时多个for语句的嵌套循环结构，每次从最外层的表中读取一行匹配查询条件的记录，传递给下一个内嵌的JOIN循环，以此类推。

下面我们给出一个例子来说明该算法的执行流程。假如我们有t1、t2和t3这3张表，需要对这3张表进行JOIN操作，并且根据条件过滤结果，则相关的JOIN语句如下：

```
SELECT * FROM t1 INNER JOIN t2 ON P1(t1, t2) INNER JOIN t3 ON P2(t2, t3) WHERE P(t1, t2, t3)
```

上面的P1(t1, t2)和P2(t2, t3)表示连接条件，P(t1, t2, t3)表示WHERE条件，其形式类似于C1(t1) AND C2(t2) AND C3(t3)。我们假设表t3没有可以利用的索引，也就是说在表t3中查询的记录必须进行全表扫描。Nest Loop Join算法处理该JOIN语句的伪码如下：

```
1. for each row in t1 match C1(t1) {
2.     for each row in t2 match P1(t1, t2) and C2(t2) {
```

```
3.        for each row in t3{  // 对表t3进行全表扫描
4.            if match P2(t2, t3) and C3(t3)
5.                r := t1|t2|t3
6.                send r to client
7.        }
8.    }
9.}
```

伪码第1行中，每次从表t1中取出一条满足WHERE查询条件的记录，我们假设该记录为t1_row，将t1_row传给内层循环。查询条件C1(t1)的严格程度（条件越严格就能够过滤掉更多的记录）将直接决定内层循环执行的次数。如果C1(t1)条件非常严格，例如是类似于WHERE id = *constant*这样的精确匹配，如果匹配的记录只有一行或者几行，那么内层循环只需要执行一次或者几次，这将会大大减少查询的执行时间。

伪码第2行用于从表t2中取出一条满足连接条件P1(t1, t2)和WHERE查询条件C2(t2)的记录，我们假设该记录为t2_row，将t1_row和t2_row传给下一个内层循环。例如，当连接条件P1(t1, t2)的形式为 t1.id = t2.id时，假如外层循环传过来的t1_row中id字段t1_row.id的值为5，此时我们会去表t2中查询id字段的值为5的记录，并且检查该记录是否满足WHERE查询条件C2(t2)，如果满足，则将其和t1_row一起传给下一个内层循环，否则检查表t2中下一条id字段的值为5的记录是否满足查询条件C2(t2)。

伪码第3到第7行中，由于表t3没有可以利用的索引，所以需要扫描表中的所有记录，对于满足连接条件P2(t2, t3)和查询条件C3(t3)的记录，将连接结果发送给客户端。其中r := t1|t2|t3表示表t1的行、表t2的行以及表t3的行组成的行记录。

图9-7给出了Nest Loop Join算法执行以上JOIN语句的流程。

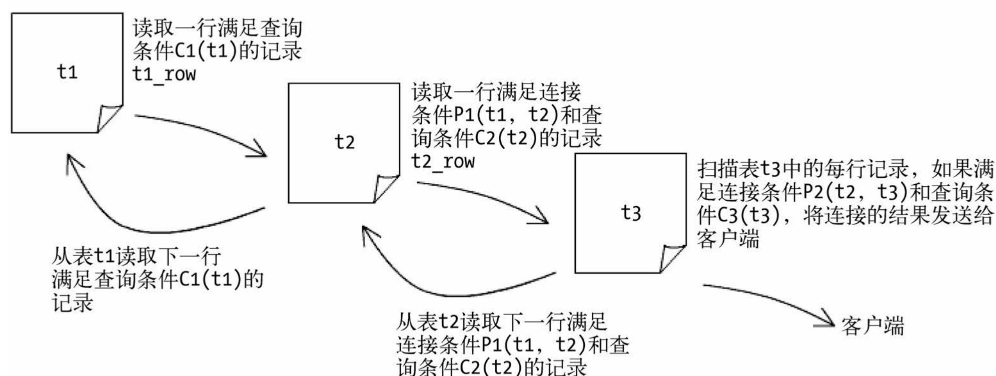


图9-7 Nest Loop Join算法

9.5.3　Block Nest Loop Join算法

由于Nest Loop Join算法每次只从外层循环读取一条记录传入到内层循环中，这会导致内层循环中表的记录被读取的次数非常多。回到上一节的例子中，假如表t1满足查询条件的记录有10条，也就是说最外层的循环需要执行10次，我们又假设对于从外层循环传入的表t1的每条记录，表t2中都有10条满足连接条件和查询条件的记录，那么可以计算出对表t3进行全表扫描（前面假设t3没有可以利用的索引）的次数为100次。

Block Nest Loop Join算法是对Nest Loop Join算法的改进，该算法使用连接缓冲区来存储连接的中间结果，以达到减少内层循环中表的读取次数的目的。我们继续使用上一节中的例子来分析Block Nest Loop Join算法的执行过程（如图9-8所示），下面给出了该算法处理例子中JOIN语句的伪代码：

```
1. for each row in t1 match C1(t1) {
2.     for each row in t2 match P1(t1, t2) and C2(t2) {
3.         store used columns from t1, t2 in join buffer
4.         if buffer is full {
5.                    for each row in t3 { // 对表t3进行全表扫描
6.                        if row match C3(t3)
7.                            for each t1, t2 combination in join buffer {
8.                                if row match P2(t2, t3) and C3(t3)
9.                                    r := t1|t2|t3
10.                                   send r to client
11.                           }
12.                   }
13.                   empty buffer
14.        }
15.    }
16.}

17.if join buffer is not empty {
18.    for each row in t3 {     // 对表t3进行全表扫描
19.        if row match C3(t3)
20.            for each t1, t2 combination in join buffer {
21.                if row match P2(t2, t3)
22.                    r := t1|t2|t3
23.                    send r to client
24.        }
25.    }
26.}
```

伪码第1行到第3行中，从表t1读取满足查询条件的记录，与表t2中满足连接条件和查询条件的记录进行连接操作，将结果存储在连接缓冲区中。

伪码第4行到第14行中，如果连接缓冲区满了，从表t3中读取一行记录，检查是否满足查询

条件，如果满足，将该记录与连接缓冲区中的每一行记录进行连接条件检查，如果满足连接条件，将它们作为连接的一条结果发送给客户端。循环这个过程，直到读取了表t3的所有记录。最后清空连接缓冲区，继续执行外层循环。

伪码第17行到第26行用于处理连接缓冲区中剩余的数据。

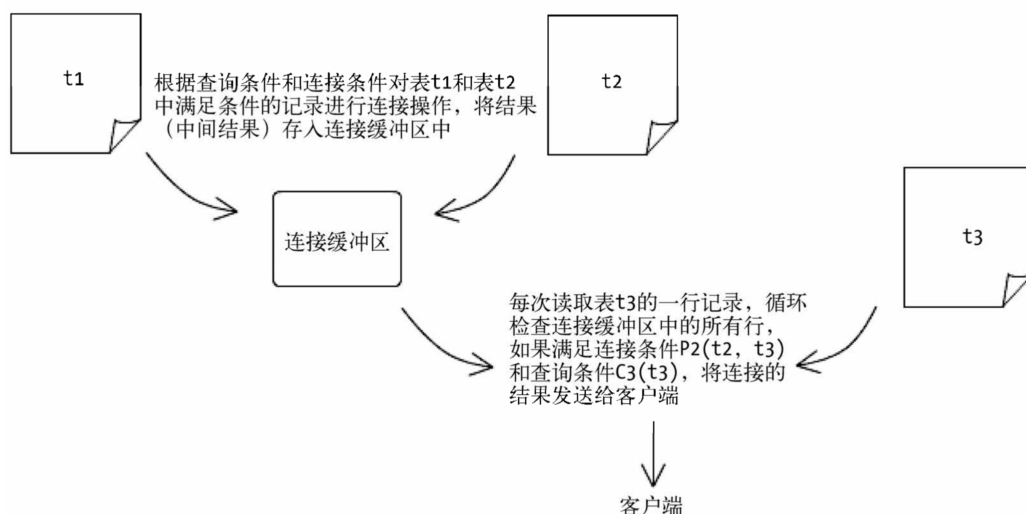


图9-8　Block Nest Loop Join算法

回到本节开始的那个例子，如果表t1包含10条满足查询条件的记录，又假设对于从外层循环传入的表t1的每条记录，表t2中都有10条满足连接条件和查询条件的记录，那么表t1和表t2进行JOIN操作的结果包含100条记录。假设连接缓冲区能够存储100条记录，那么只需要对表t3进行一次全表扫描就能够得到最终的结果。与Nest Loop Join算法扫描100遍相比，这个算法的效率得到了明显的提升。

在MySQL上，我们可以使用SELECT @@optimizer_switch命令来查看Block Nest Loop Join算法是否被开启，其中block_nested_loop=on表示Block Nest Loop Join算法被启用：

```
mysql> SELECT @@optimizer_switch;
+-------------------------------------------------------------------------------------------+
| @@optimizer_switch                                                                        |
+-------------------------------------------------------------------------------------------+
index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_merge_intersection=on,engine_c
ondition_pushdown=on,index_condition_pushdown=on,mrr=on,mrr_cost_based=on,block_nested_loop=on,bat
ched_key_access=off,materialization=on,semijoin=on,loosescan=on,firstmatch=on,subquery_materializa
tion_cost_based=on,use_index_extensions=on
+-------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

9

此外，通过设置参数join_buffer_size的值可以指定连接缓冲区的大小。假设S为连接缓冲区中一条记录的大小（表t1的行和表t2的行连接后的大小），而C是表t1和表t2 连接结果的总条数，那么表t3被扫描的次数为(S * C) / join_buffer_size + 1。我们可以看到，当连接缓冲区增大之后，会减少表t3的扫描次数，当join_buffer_size增加到大于S * C的大小后，就不能减少表t3的扫描次数了。

```
mysql> SHOW VARIABLES LIKE "join_buffer_size";
+---------------------------+-------------------+
| Variable_name             | Value             |
+---------------------------+-------------------+
| join_buffer_size          | 262144            |
+---------------------------+-------------------+
1 row in set (0.00 sec)
```

9.5.4 Batched Key Access Join算法

MySQL/MariaDB通常使用Nest Loop Join算法来对两张表或者多张表进行JOIN操作，也就是说每次从第一张表读取一条满足查询条件的记录，根据连接条件在第二张表上使用索引（如果存在可用的索引）查询匹配的记录。这种情况下，如果第一张表有1000条满足查询条件的记录，那么会在第二张表上执行1000次查询操作，这1000次查询的键值很可能是随机的，当第二张表的数据没有完全在内存中时，这1000次查询将会导致多次随机I/O的发生，从而导致性能下降。

MySQL 5.6和MariaDB 5.5针对这种情况对原有的连接算法进行了优化，优化后的算法每次从第一张表中读取多条满足查询条件的记录放入缓冲区中，然后将这些记录的键值进行打包，根据连接条件去第二张表中查询。这样存储引擎就会使用MRR（Multi Range Read）接口来处理该请求，减少了随机I/O的发生。MariaDB还做了更深一步的优化，在打包键值进行查询时会对这些键值进行排序。这种优化后的算法叫做Batched Key Access Join算法。

9.5.5 Hash Join算法

MySQL只支持前面介绍的Nest Loop Join、Block Nest Loop Join和Batched Key Access Join这几种算法，而MariaDB除了支持这几种算法外，还为等值连接引入了一种新的JOIN算法——Hash Join算法。

Hash Join算法主要分为两步：创建哈希表以及在哈希表中查找。下面我们介绍下该算法的具体执行流程。

假设我们要对表A和表B这两个表进行JOIN操作，则Hash Join算法首先选取一张数据量比较小的表（例如表A），在内存中建立一张哈希表，将表A的记录映射到该哈希表内，然后遍历表B中的每条记录，去哈希表中查询，看是否有匹配连接条件的记录，如图9-9所示。

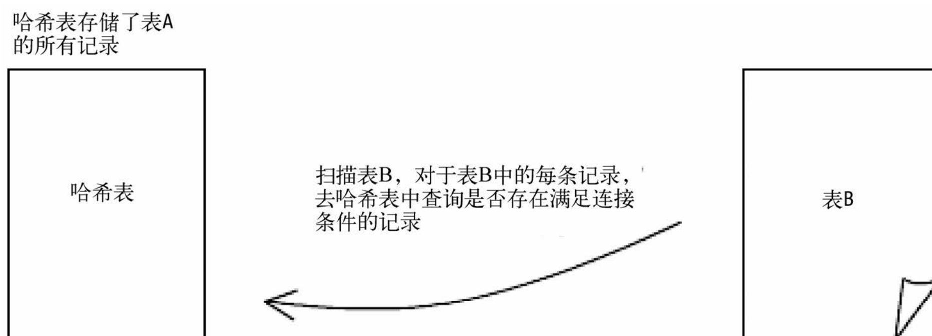


图9-9 Hash Join算法

Hash Join算法比较适合在以下几种场景下使用：

- 一张小表和一张大表进行连接操作；
- 没有可以利用的索引；
- 连接结果集比较大。

9.5.6 Sort Merge Join算法

Sort Merge Join算法是另一种比较流行的连接算法，该算法主要由以下几个步骤组成。

(1) 扫描所有参与连接的表。
(2) 对第(1)步扫描的结果进行排序。
(3) 进行归并连接。

在归并的过程中，从最小值开始同时遍历两个排好序的集合，当遇到A集合的某个子集和B集合的子集相等时，计算它们的笛卡尔积，并将其作为输出结果，当遍历完两个集合或者遍历完了一个集合并且另一个集合剩余的元素都大于第一个集合的最大值时，结束归并过程。图9-10给出了对已排序的两个集合做归并连接的过程。

Sort Merge Join算法最大的开销在于对数据集进行排序，如果数据集本来就是有序的，该算法比其他的连接算法会有一定的优势。

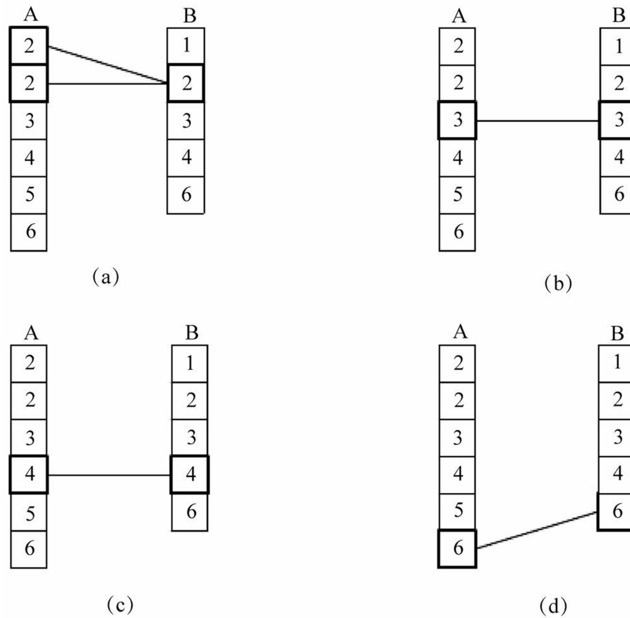


图9-10　归并过程

9.6　小结

本章中，我们介绍了MariaDB/MySQL中使用的一些数据结构以及复杂查询ORDER BY和JOIN的算法实现。下面我们对这些内容做个简短的回顾。

- B+树是为磁盘或其他直接存取辅助存储设备而设计的一种平衡查找树。B+树与红黑树和其他平衡二叉树类似，但在降低磁盘的I/O次数方面做得更好。B+树在对磁盘I/O非常敏感的系统中得到了非常广泛的应用，例如文件系统和数据库系统。
- 通常情况下，当收到ORDER BY请求时，MariaDB/MySQL采用filesort算法对结果集按照指定的字段进行排序，然后返回给客户端；当满足某种条件时，MariaDB/MySQL会利用数据库的索引（这里指的是B+树类型的索引）具有一定的顺序这一特性，按照索引的顺序将结果返回给客户端，这样就省略了对结果集的排序过程，加快了查询的响应速度。
- MySQL只支持Nest Loop Join、Block Nest Loop Join和Batched Key Access Join这几种算法。Nest Loop Join算法每次从最外层的表中读取一行匹配查询条件的记录，传递给下一个内嵌的JOIN循环，以此类推。Block Nest Loop Join算法是对Nest Loop Join算法的改进，该算法使用连接缓冲区来存储连接的中间结果，以达到减少内层循环中表的读取次数的目的。

MariaDB除了支持上面提到的几种算法外，还支持Hash Join算法，该算法通过对表记录建立哈希表，提高连接过程中的查询速度。

第 10 章

分布式数据库

随着数据库应用的不断发展，数据的规模也在不断扩大，传统的集中式数据库在容量、性能、可扩展性等方面都遇到了问题，分布式数据库随之产生。分布式数据库利用计算机网络将物理上分散的多个数据库单元连接起来形成的一个逻辑上统一的数据库。

本章的内容主要包括：

- 分布式数据库概要
- 数据的分片方式
- 分布式数据库实践——京东分布式数据库架构

10.1 分布式数据库概要

本节中，我们先介绍分布式数据库的概念及其特点，然后介绍分布式数据库中的一些技术难点。

10.1.1 分布式数据库的特点

分布式数据库是数据库系统与计算机网络系统相结合的产物，具有以下几个鲜明的特点。

- **物理分布性**。分布式数据库的数据不是存储在一个结点上，而是分散存储在由计算机网络连接起来的多个结点上，而这对于数据库的使用者来说是不可见的。
- **逻辑整体性**。虽然分布式数据库的数据物理上是分布在不同的结点上，但这些分散的数据在逻辑上确是一个整体，物理的分布性对于用户来说是不可见的。在用户看来，使用分布式数据库和使用集中式数据库系统是完全一样的，没有任何区别。

分布式数据库系统是在集中式数据库系统的基础上发展而来的，与集中式数据库系统相比，具有以下几个优点。

- **较大的灵活性和可扩展性**。在分布式数据库系统中，容量和系统处理能力的扩充是非常简单方便的。

- **更好的经济性**。与一个使用大型计算机支持一个大型的数据库系统相比，使用廉价的普通机器组成一个分布式的数据库往往具有更高的性价比。
- **物理表更小，真正的并行，效率更好**。在分布式数据库中执行一条查询语句时，该语句会被拆分成多个子查询分发到多台物理机器上，同时并发执行。与在一张大表中查询数据相比，我们查询的表的数据量更小，并且查询是在多台机器上并发执行的。

10.1.2　系统的扩展方式

通常，系统的扩展方式包括单机垂直扩展（scale-up）和多机水平扩展（scale-out）两种，这两种方式都有自己的优点和缺点，具体如下所示。

1. 单机垂直扩展

所谓单机垂直扩展，就是购买或更换更加高端的机器，使用更快的存储设备，提高单台机器的性能。

这种方式的扩展具有一个明显优点：业务不用修改代码。因为仅仅是使用更好的硬件来提高单台数据库的性能，对于业务来说没有任何的改变。

同时，单机垂直扩展存在以下几个明显的缺点。

- **成本巨大**。通常，性能更好的机器或者存储比普通的服务器或存储在价格上会高出很多。
- **扩展性不好**。在进行单机垂直扩展之后，虽然能够满足当前的需求，但随着业务的增长，不久的将来又会面临系统到达瓶颈的问题。

2. 多机水平扩展

所谓多机水平扩展，就是通过添加廉价的普通机器到原有的系统中来提高原有系统的容量和处理能力。这种方式的扩展具有如下几个优点。

- **成本小**。由于我们是通过添加廉价的普通机器到原有的系统中来扩展原有系统的能力，所以这种方式具有常量的边际成本。
- **较好的扩展能力**。理论上，这种方式的扩展是没有限制的，不需要担心会再次遇到系统瓶颈的问题。

同时，这种方式的扩展也具有几个缺点。

- **第一次重构需要付出成本**。当我们从单机系统扩展到分布式的多机系统时，需要做一些工作，使分布式系统具有和单台系统同样的功能。
- **完成相同功能比单机扩展付出的成本更高**。在分布式多机系统中实现一个功能，不仅要保证该功能的局部可用性和一致性，同时需要保证这一功能的全局可用性和一致性。

10.1.3 分布式数据库中的技术难点

分布式数据库在物理上是分散的，而在逻辑上却是统一的，这一特性决定了在实现分布式数据库系统时将会面临一些挑战。

- **分布式数据库的查询处理**。分布式数据库需要向用户提供一个统一的数据访问接口。对于用户来说，使用分布式数据库和集中式数据库没有任何区别，好像所有的数据都存储在单台机器上一样。但是，实际数据是分布在不同的机器上，这使得在查询处理的过程中需要在各个结点间进行通信，需要对各个结点的查询结果进行收集甚至进行进一步运算。
- **分布式事务**。由于在分布式数据库中逻辑上的一个表可能由分布在不同机器上的多个物理表组成。分布式事务不仅要保证在单个结点上事务的ACID特性，而且要保证全局的ACID特性。对于分布式数据库来说，分布式事务是一个难解决的问题，需要进行复杂的一致性和完整性校验。

10.2 数据的分片方式

数据分片也叫数据分割，是分布式数据库的特征之一。在分布式数据库中，分散的物理表是通过全局的逻辑表按照某种方式分割而来的。数据的分片方式主要包括水平分片、垂直分片和混合分片3种，如图10-1所示。

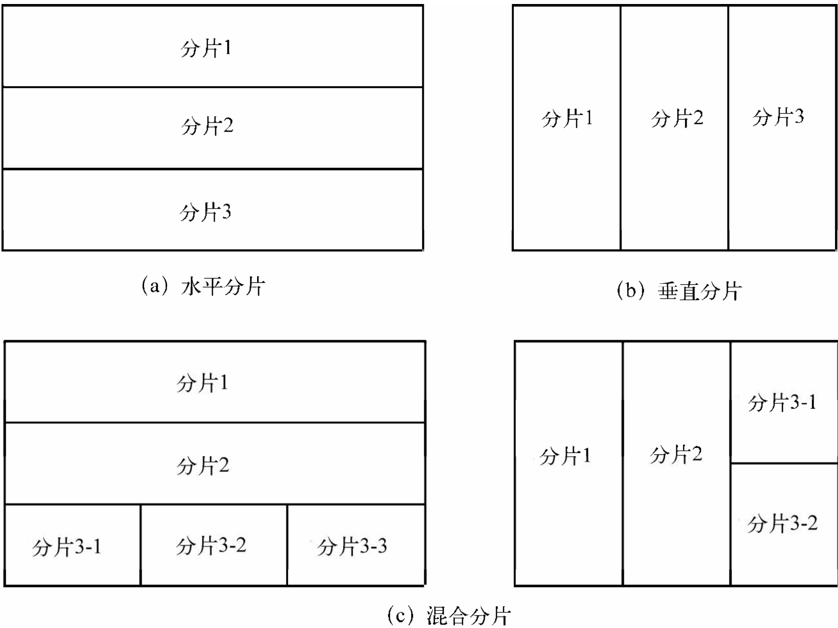


图10-1 数据分片方式

- **水平分片**。也叫按行分片，将关系r按行分为若干子集r1,r2,...rn，其中每个子集称为一个分片。水平分片通常在数据量非常大的时候使用，这种方式的分片没有任何的数据冗余。
- **垂直分片**。也叫按列分片，将关系r按列分为若干属性子集，每个子集称为一个垂直分片。这种分片方式比较适用于数据库中表的属性特别多，也就是包含的列特别多，而且我们的查询只需要返回很少的部分属性时。垂直分片需要在所有的子表中都记录主键信息，也就是说这种分片方式会产生一定的数据冗余。
- **混合分片**。这种分片方式对数据按照某种分片方式分片之后，再对数据子集按照另一种分片方式继续分片。

10.3 分布式数据库实践——京东分布式数据库系统

随着业务的高速发展，京东对基础研发的依赖也变得越来越强烈，京东分布式数据库就是众多基础设施中的一项。MySQL作为一款优秀的开源的数据库管理系统，在各大互联网公司应用得非常广泛，但一直以来官方并没有提供一套真正成熟的分布式数据库系统的解决方案，所以各个公司在面临自身的业务需求时，实现的分布式数据库系统也各不相同。

分布式数据库实现方案总体可以划分为两类。一类是客户端的方案，在使用这类方案时需要应用程序使用专门开发的客户端。对于一个已有的应用程序来说，要使用这类分布式数据库方案，可能会涉及一些代码的改动甚至是一些程序逻辑上的调整。另一类是代理（proxy）的方案，在使用这类方案时，应用程序不需要做改动，可以直接使用原生的客户端，但是性能上可能会打折扣。因为数据包从客户端到代理结点，然后再从代理结点再到MySQL，中间多了一次网络IO。鉴于京东的业务现状，我们采用了代理方案，让现有的应用程序尽可能少做改动，以达到快速将现有业务切换到分布式数据库系统上的目的。

10.3.1 京东分布式数据库系统架构

京东分布式数据库系统的整体架构如图10-2所示，整个系统可以提供高可用、高可靠、可扩展且全程运维自动化的服务。

整个系统主要包括以下几个模块。

- **Jproxy（代理结点）模块**：MySQL的代理层，兼容MySQL协议。应用程序的访问请求首先都会发送给Jproxy，Jproxy会根据路由信息对SQL语句进行拆分处理，将拆分后的SQL语句发送给后端各个MySQL实例，然后Jproxy会将各个MySQL返回的结果进行合并处理，最终返回给应用程序。
- **dbAgent模块**：每个MySQL实例对应一个dbAgent，两者部署在同一台机器上，负责MySQL实例的启动、停止、存活、健康状况监控以及数据的高可靠备份等相关管理工作。

图10-2 京东分布式数据库系统的架构

- Failover模块（异常处理模块）：负责各个高可用组（HA group）的故障切换、MySQL实例之间关系的初始化以及后续维护调整等。
- Transfer模块（迁移模块）：负责对实例上的数据进行迁移。
- InitAgent模块（初始化模块）：主要用于初始化环境，包括各种依赖的初始化。
- Collector模块（信息采集模块）：接收dbAgent采集的各个实例的使用状况，例如I/O压力、磁盘空间等。
- Web API模块：负责提供各种运维管理相关的API。
- Manager模块（中心管理模块）：这是整个系统的核心决策模块，负责维护路由、故障切换的决策、迁移的控制及其他各种调度。

另外，整个系统使用了消息队列将一些耗时操作异步化。此外，它还使用ZooKeeper对MySQL实例的存活进行监控。

10.3.2 高可用组的初始化

一个高可用组是指由一个主库和多个从库构成的集合，而整个分布式数据库系统由多个高可

用组构成。当需要往该系统中添加高可用组时，可以通过Web API往元信息数据库中添加机器相关的信息，然后请求创建相关的高可用组。Manager通过消息队列获取到创建高可用组的请求以后，再到元信息数据库中获取相关的信息。接着，通知InitAgent初始化相关机器的环境，包括磁盘的格式化、物理卷和逻辑卷的创建以及各种依赖包的下载，比如dbAgent、MySQL以及Python环境等。

当机器环境部署好后，InitAgent会告诉Manager，接着Manager通知Failover模块去启动相关的MySQL实例并设置相关的主从关系，如图10-3所示。高可用组的初始化本身从某种意义上来说也是一次特殊的Failover处理。Failover模块会通知dbAgent将相关的MySQL实例启动起来，设置好虚拟IP，反馈给Manager说该高可用组已经启动。在高可用组的整个初始化过程中，Manager会将各个状态写入元信息数据库中，所以我们可以明确地知道一个高可用组创建进行到了哪一步。

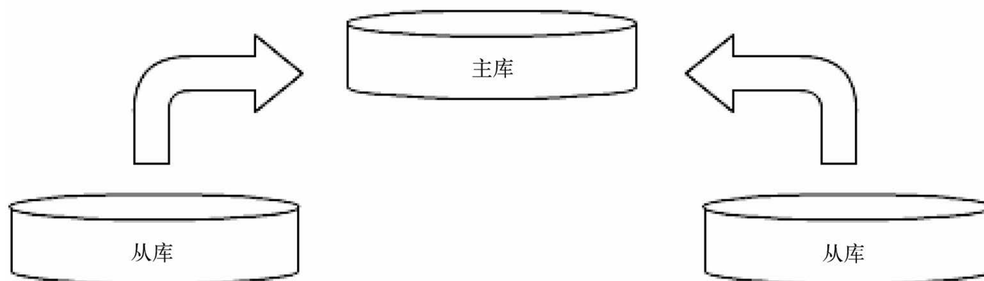


图10-3　高可用组

10.3.3　数据的分片

当高可用组部署到位以后，应用开始接入。我们首先需要对业务的库表做一些分析，确认每张表是否需要分片，如果需要，则需要知道分片的主键是什么以及需要分几片。图10-4所示是该分布式数据库系统的分片示意图，其中schema表示应用使用的数据库，也就是一个逻辑上的数据库，bucket表示一个分片对应的桶，高可用组表示一个高可用的MySQL主从集合。针对schema中的table1和table2进行分析，其中table1不做分片处理，全部落在bucket1这个桶上，table2需要做分片处理，依次落在bucket1、bucket2、bucket3和bucket4这 4 个桶上。

当业务需求分析清楚以后，确认分片主键以及分片的个数，然后通过Manager到相应高可用组的各个实例上创建好相关的库表结构，并将路由存到元信息数据库中，最后将路由推给各个Jproxy。在库表结构以及路由关系配置好以后，应用就可以用原生的MySQL客户端连接上Jproxy进行使用，接下来所有的操作就像是操作单个MySQL一样。

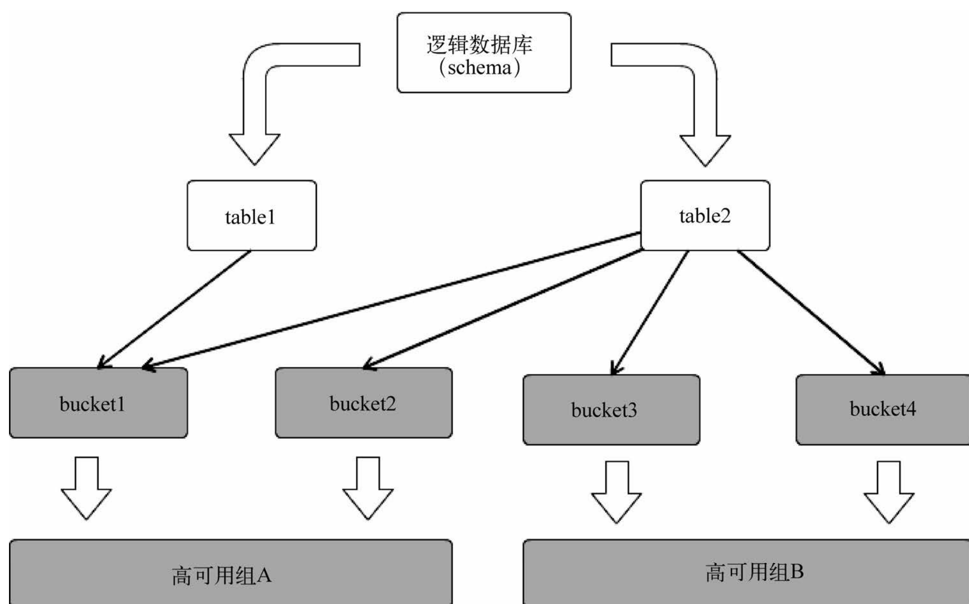


图10-4　数据的分片

10.3.4　系统的高可用性

每个MySQL 高可用组都由一个主库和多个从库组成，每个MySQL实例所在的机器上都会有一个对应的dbAgent。dbAgent会监控每个实例的健康状况，并将存活信息反馈给ZooKeeper，Manager模块会通过ZooKeeper获取到每个高可用组中每个MySQL实例的存活状况，如图10-5所示。

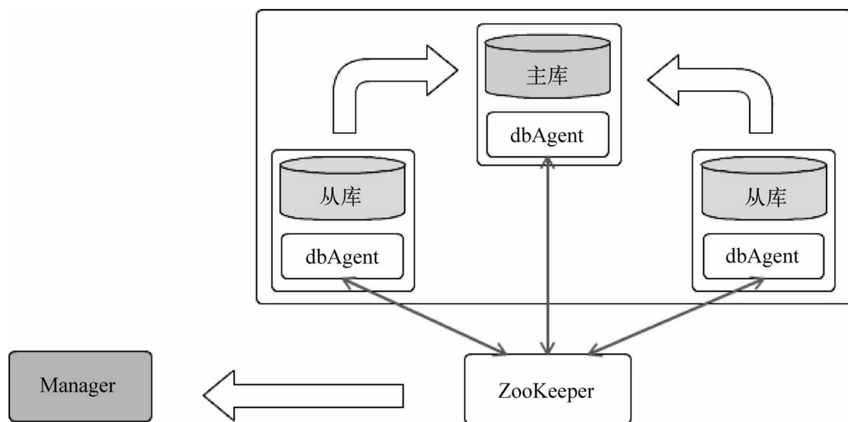


图10-5　MySQL实例存活监控

如果Manager通过ZooKeeper发现主库可能已经死亡，此时Manager会再次确认主库实例是否

是真的已经死亡。当Manager判定主库为死亡状态时，Manager会告知Failover模块去做Failover处理。Failover模块将通过与dbAgent交互，将某个slave提升为master，然后将其他的slave作为该新提升的master的slave。等Failover模块处理成功以后，Manager会更新路由信息，并将新的路由信息推送给所有的Jproxy。整个过程中的每一步中间状态都会记录到元信息数据库中。

Failover模块可以根据每个高可用组创建时人为指定的机器优先级（例如以机器性能为依据等）来选择将哪个slave优先提升为master。但是人为指定的机器上的MySQL实例未必拥有最新的数据，有可能是落后于其他slave实例的，此时需要将所有的slave都与原master显式地断开，然后将待提升的slave依次作为其他slave实例的slave，从而保证待提升的slave上拥有最新的数据。然后再将该slave提升为master角色，最后将其他slave都设置为该实例的slave。主从之间的复制是基于GTID的，所以整个过程是相对简单可控的。如果高可用组在创建的时候没有人为指定机器优先级，Failover模块可以从高可用组中选取一个拥有最新数据的从库作为主库。因为每个高可用组都会有一个对应的虚拟IP，所以当新的slave真正提升为master以后，Failover模块还需要再做一些虚拟IP的切换工作。图10-6所示给出了故障切换后的状态。

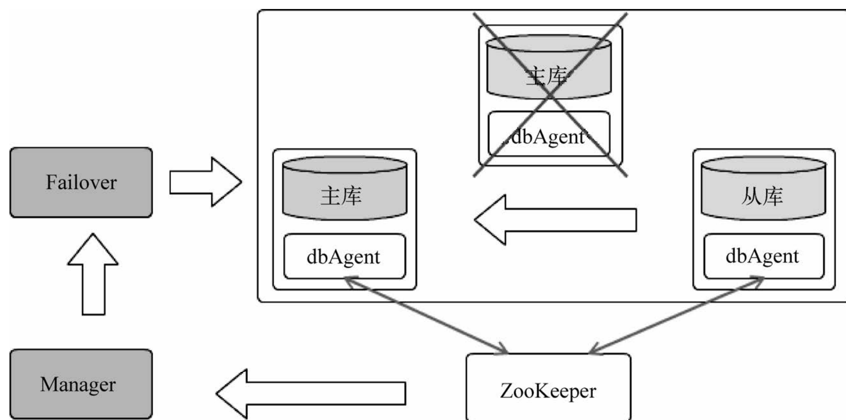


图10-6　Failover处理过程

高可用除了需要保证异常情况下服务高可用外，还需要提供正常计划之中的机器停机维护的服务高可用。例如，当某台机器因为硬件有异常而存在风险需要做下线处理时，如果该机器上的MySQL实例是slave角色，则相对比较容易处理：Manager会通知Failover模块将对应的实例停止，更新路由信息并告知Jproxy，相应的机器便可以下线了。但如果需要停机的机器上的MySQL实例是master角色，则需要先通知Jproxy表示需要做一次人工切换，使涉及的所有逻辑数据库仅提供只读服务，然后再按照类似master出异常的流程来处理，从多个slave中选出一个slave提升为master，然后把其他的slave作为新的master的slave，更新路由之后再告知Jproxy，使之前涉及的逻辑数据库恢复读写服务，最后再将该机器下线。

除了将机器下线以外，还需要能支持向高可用组中添加新机器的需求。如果需要往高可用组

中添加一个新的实例，Manager首先会通过InitAgent到新添加的机器上初始化好相关环境，之后Manager再通知Failover模块去启动MySQL实例以及设置好对应的主从关系。

上述几种场景不管是正常情况还是异常情况，除添加一个从库的情况外，其他都属于主从切换的情况，但仅仅提供上述支持还是不够的，因为线上很多时候可能会出现主从实例本身都是存活的情况，但是主从复制却延迟了，此时需要人工介入处理。如图10-7所示，每个dbAgent会采集对应实例的状态信息，然后上传给Collector模块，Collector会对各个实例的状态信息进行整理汇总，然后将一些重要的信息存入元信息数据库中，包括主从复制延迟等信息。Web API模块会有定时任务扫描元信息数据库中的记录，同时会提供相应的API注册到京东的统一监控系统中，如果发现有异常，统一监控系统会通知相应的负责人介入处理。

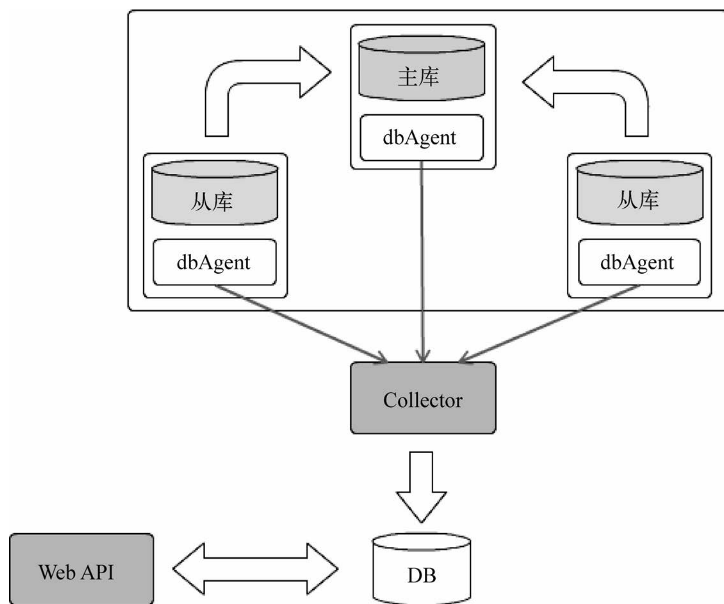


图10-7　监控状况监控

10.3.5　系统的可扩展性

对于一个分布式系统来说，除了保证高可用性以外，还需要提供良好的可扩展性。前面提到过，系统的扩展主要包括垂直扩展和水平扩展两种。垂直扩展一般是指当某台机器达到瓶颈时，例如单台机器8TB的硬盘空间不够了，就考虑增加到16TB，如果16核的CPU是瓶颈，就考虑改用32核的CPU，简单地说就是通过改善单机的性能来应对业务上的压力。对于很多业务来说，采用垂直扩展或许可以支撑一段时间，但是对于规模比较大，特别是增长比较明显的业务，垂直扩展并不能解决根本问题，此时就需要考虑采用水平扩展方式了。水平扩展一般是指当系统遇到瓶颈的时候，通过往系统中添加新的普通机器来提高系统的支撑能力，如果业务后续继续增长，就继

续添加新的机器。所以对于一个分布式系统来说，支持水平扩展的扩展方式更符合可持续发展的思想。

要支持水平扩展的扩展方式，分布式数据库系统首先需要支持分片，在这个基础上支持迁移，最后是支持分片的拆分。分片在之前章节中已经介绍过了，一个schema中的表会被拆分到多个bucket中，每个bucket就是一个分片。在应用刚接入的时候，虽然分了多个bucket，但是可能数据量比较小，为了充分利用硬件资源，我们通常会将把这些bucket放在一个或者少数几个的MySQL实例上。随着应用数据的进一步增加，当单台MySQL实例所在机器上的磁盘较满或者因为负载较高导致响应较慢时，就可以考虑将该实例中的bucket迁移到新的MySQL实例中。在图10-8所示的迁移示意图中，一个应用分为4片，分别为bucket1、bucket2、bucket3和bucket4，当bucket1和bucket2所在的高可用组中的实例磁盘空间剩余较少时，可以往整个系统中添加一个新的高可用组，然后将bucket2从之前的高可用组中迁移到新的高可用组中。这种方式已经可以支撑绝大多数业务在很长时间内的持续发展了。例如，一个新的应用接入时，预分片的时候划分为128片，在一开始的时候，这128片可能只是落在4个高可用组上，当数据量不断增大以后，可以不断地将分片进行迁移，做到每一片对应一个高可用组，最终落到128个高可用组上。

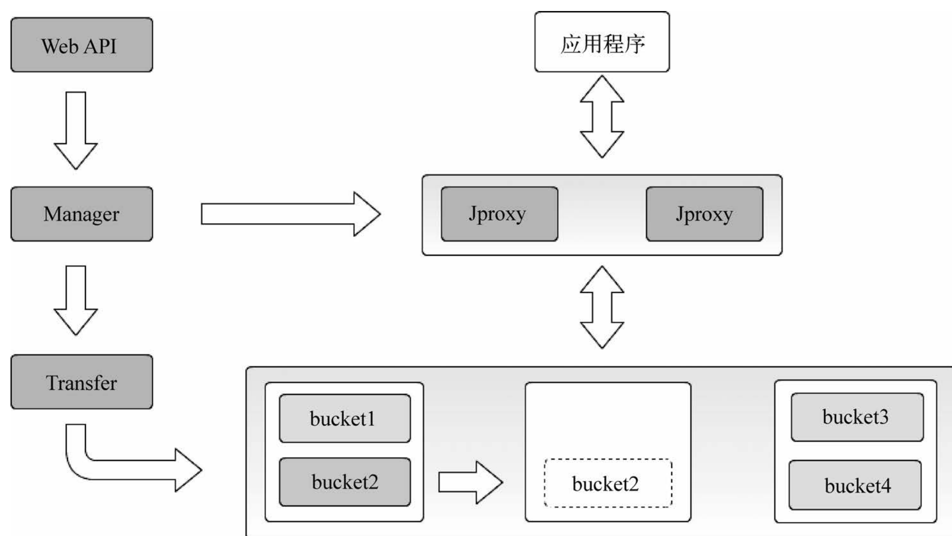


图10-8　迁移示意图

除了简单地将分片从一个高可用组迁移到另一个高可用组以外，有些特殊的应用可能还需要将已经分片的数据继续进行拆分，例如一个应用原本分了16片，经过几次迁移以后，16片已经落在了16个高可用组上，但是数据增长的还是很快，此时就需要考虑将原有的分片进行拆分。分片拆分和分片迁移类似，但略有不同，后者是将分片数据从一个高可用组迁移到另一个高可用组，前者是将分片数据从一个高可用组拆分到两个高可用组。分片拆分需要对数据进行一次计算处理，确认每条记录应该位于哪个分片，所以相对会复杂一点。在图10-9所示的分片拆分示意图中，

当一个高可用组的磁盘已经满了，此时需要将bucket2拆分成两片bucket2-1和bucket2-2，此时整个应用由原来的四片拆分成了五片，分别为bucket1、bucket2-1、bucket2-2、bucket3和bucket4。

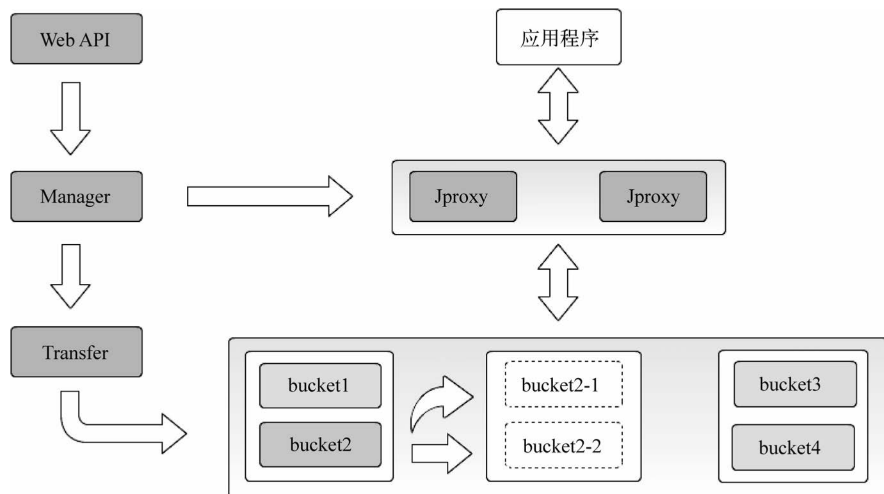


图10-9 分片拆分

不管是分片迁移还是分片拆分，本质都是数据迁移，京东分布式数据库系统的迁移工作是完全自动化在线迁移的，主要由Transfer模块完成。提交了迁移任务以后，Transfer会在迁移任务指定的时间开始数据迁移，具体的步骤如图10-10所示。因为每一个高可用组中都有一个master和多个slave，考虑到实现的难易程度，整个迁移过程都选择对高可用组中的master实例进行操作。首先到源实例上将数据导出来，然后将导出来的数据恢复到目标实例，如果是分片拆分的话，恢复的过程需要有一次解析计算，确保数据落到正确的分片上。当恢复完以后，需要对数据进行一次校验，确认此时数据没有出现异常。这些步骤都是在线下操作的，对业务是完全不可见的，所以在此期间可能会有新的数据进入待迁移的分片中，此时需要将这些增量数据抽取出来，然后恢复到目标实例上。在分片拆分的时候，同样需要对增量数据进行计算，确保数据落到正确的分片上。

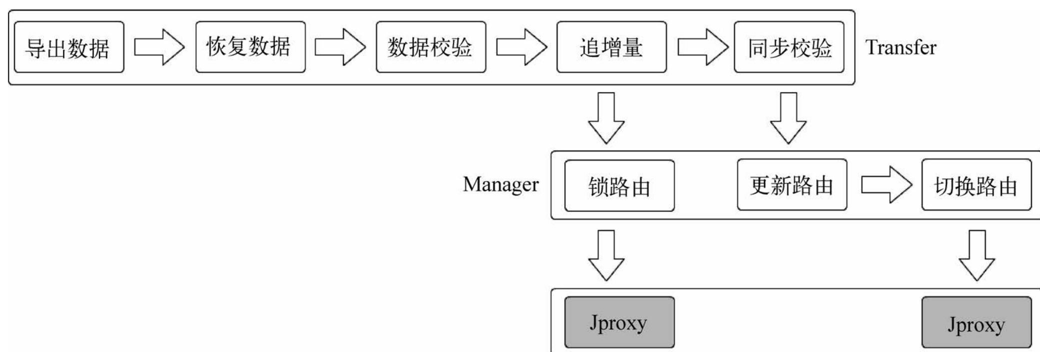


图10-10 迁移流程

当追增量追到一定程度时，需要请求Manager将Jproxy的路由锁住，此时该分片对应的schema仅提供只读服务，其他不涉及迁移的schema的访问都是正常的。如果路由不锁住的话，增量可能永远追不完。但是考虑到大部分数据已经迁移完毕，所以锁路由的时间会非常短暂。线上实际迁移的时候，一个拥有几十GB数据的bucket迁移，锁路由时间只有短短的几秒钟。

当锁住路由以后，我们会将剩余的少量增量完全追完，最后再做一次同步校验，确保增量追赶的正确性。此时Transfer模块会请求Manager模块更新路由，Manager会将路由更新以后，将其存入元信息数据库中，接着再将新的路由推送给所有的Jproxy，接着将所有的Jproxy路由解锁，至此路由切换完成，整个迁移过程结束。

整个自动化在线迁移系统还是比较复杂的，还有很多其他因素需要考虑，例如各种并发情况、迁移时间点的选择以及迁移过程中发生异常主从切换，等等。

10.4　小结

本章中，我们首先介绍了分布式数据库的一些概念、数据的分片方式以及分布式数据库实现中的一些技术难点，等等。

最后，我们重点介绍了京东的分布式数据库系统，介绍了整体的系统架构以及高可用性、可扩展性和高可靠性的实现。整个设计思想是中心化管理，所有的调度以及路由维护等都由Manager管理。分布式系统是比较复杂的，需要处理的细节也比较多。有单点故障的节点都需要通过主从备份来消除单点故障，有并发处理的场景在设计时都需要尽可能地考虑清楚，最后再加上充分的测试以及上线后完善的监控，最终保证整个系统稳定运行。

附录 A

数据库与IO资源控制



这一部分内容主要是延伸阅读。通常，当我们的应用数比较多的时候，可能会将不同的应用部署在同一台物理服务器上（通常是不同类型的应用，例如将IO密集型应用和计算密集型应用放在一台物理机上，以达到充分利用物理服务器资源的目的），这个时候就会涉及各个应用之间的隔离问题以及资源控制问题。容器方案随之产生了，现在，很多容器方案（比如docker和lxc）中，资源控制和隔离的本质都是利用Linux内核的CGroup和namespace机制来实现的。

本章中，我们将对数据库的资源控制进行相关的介绍，对Linux资源控制系统CGroup进行细致的讲解。

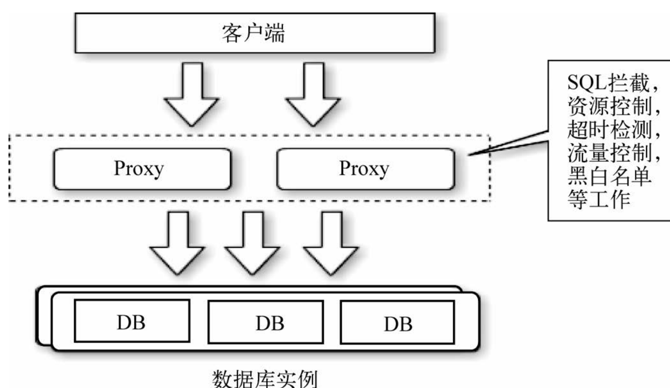
A.1　数据库资源控制方案简介

云数据库根据用户或应用类型的不同，按照资源部署情况，大体可以分为共享型数据库和独享型数据库。针对数据量和访问量都比较大的应用，会分配独立的数据库实例和相应的物理资源。针对大多数的小应用可能是多个应用共用同一数据库实例，并且部署在同一台物理设备资源上，那么如何做到多个应用之间互不影响就显得尤为重要了。具体的隔离方案可以在应用层来做，也可以在OS层来做，或者是使用虚拟机的方案，我们先简单介绍下主要的几种方案。

A.1.1　基于应用层的资源控制

基于应用层的隔离方案如图A-1所示，基本是通过在数据库访问接入层上引入一些策略来保障多个应用之间尽量做到互不影响，例如针对低效SQL语句拦截、针对应用级别进行流量控制，等等。

这种隔离方案总体上比较简单，但由于其所作用的层次比较高，所以基本无法做到细粒度相对准确的一个资源控制。基于SQL的拦截总体上也只是一个预判，流量控制上可能只能控制网络带宽的分配，后端DB资源的控制及相应IO资源的控制可能就无法精确掌控。



图A-1　基于应用层的资源控制

A.1.2　基于OS层的资源控制

基于OS层的资源控制目前主要是使用kernel的CGroup机制。CGroup机制通过VFS暴露的文件系统接口来使用户能够控制不同进程对操作系统资源的使用，主要包括CPU、内存、存储、网络等方面。由于CGroup位于内核层，所以对资源控制的粒度就相对精细很多。不过CGroup本身只是一套框架，具体功能要看各个内核子系统维护者的实现程度，所以很多特性可能需要比较新的内核版本。并且CGroup各个内核子系统每天还在不断发生变化，这点用户可能需要单独考虑和权衡。

A.2　CGroup的基本使用

本节中，我们将介绍CGroup的一些基本用法。

A.2.1　内核选项检查及编译

要使用CGroup，首先需要确认我们的内核对CGroup这个子系统的支持程度。对于不同的内核版本，有些CGroup选项默认是关闭的，需要根据使用需要自行打开并重新编译内核。

在内核源代码包的根目录下执行如下命令：

```
root# make menuconfig
```

在弹出的内核配置菜单中选择 General setup → Control Group support 菜单项，从打开的界面中确认图A-2中的这些子系统是否已经被打开。

```
-*- Control Group support
[*]   Example debug cgroup subsystem
[*]   Freezer cgroup subsystem
[*]   Device controller for cgroups
[*]   Cpuset support
[*]     Include legacy /proc/<pid>/cpuset file
[*]   Simple CPU accounting cgroup subsystem
[*]   Resource counters
[*]     Memory Resource Controller for Control Groups
[*]       Memory Resource Controller Swap Extension
[*]         Memory Resource Controller Swap Extension enabled by default
[*]       Memory Resource Controller Kernel Memory accounting
[*]     HugeTLB Resource Controller for Control Groups
[*]   Enable perf_event per-cpu per-container group (cgroup) monitoring
-*-   Group CPU scheduler  --->
[*]   Block IO controller
[*]     Enable Block IO controller debugging
```

图A-2　检查CGroup子系统是否打开

进入Group CPU scheduler →子菜单，确认CPU调度器相关的内容已经被打开，如图A-3所示。

```
-*- Group CPU scheduler
-*-   Group scheduling for SCHED_OTHER
[*]     CPU bandwidth provisioning for FAIR_GROUP_SCHED
[*]   Group scheduling for SCHED_RR/FIFO
```

图A-3　确认CPU调度器相关的内容已经打开

如果默认情况满足需求，就无须做其他工作。如果默认情况不能满足需求，则将相应选项打开后，重新编译内核，具体过程如下。

保存并退出刚刚配置好的menuconfig菜单后，执行如下命令：

```
root# make & make modules_install & make install
```

在编译安装过程中，会自动设置好grub等启动信息，之后重启操作系统，进入grub菜单，选择新编译的内核版本即可。

A.2.2　CGroup的操作方法

CGroup是通过内核的VFS接口暴露给用户使用的。也就是说，它本身是作为一个文件系统来完成用户数据到内核的传递的。所以，使用CGroup时，首先需要挂载CGroup文件系统（如图A-4所示），具体使用方式如下：

```
root# mount -t cgroup <dev name> <mount dir>
```

```
root@test:~# mount -t cgroup cg /sys/fs/cgroup
root@test:~# ls /sys/fs/cgroup/
blkio.io_merged                    blkio.io_wait_time                blkio.throttle.read_iops_device   cgroup.sane_behavior   cpuset.mem_exclusive              cpu.shares
blkio.io_merged_recursive          blkio.io_wait_time_recursive      blkio.throttle.write_bps_device   cpuacct.stat           cpuset.mem_hardwall               cpu.stat
blkio.io_queued                    blkio.leaf_weight                 blkio.throttle.write_iops_device  cpuacct.usage          cpuset.memory_migrate             devices.allow
blkio.io_queued_recursive          blkio.leaf_weight_device          blkio.time                        cpuacct.usage_percpu   cpuset.memory_pressure            devices.deny
blkio.io_service_bytes             blkio.reset_stats                 blkio.time_recursive              cpu.cfs_period_us      cpuset.memory_pressure_enabled    devices.list
blkio.io_service_bytes_recursive   blkio.sectors                     blkio.weight                      cpu.cfs_quota_us       cpuset.memory_spread_page         notify_on_release
blkio.io_serviced                  blkio.sectors_recursive           blkio.weight_device               cpu.rt_period_us       cpuset.memory_spread_slab         release_agent
blkio.io_serviced_recursive        blkio.throttle.io_service_bytes   cgroup.clone_children             cpu.rt_runtime_us      cpuset.mems                       tasks
blkio.io_service_time              blkio.throttle.io_serviced        cgroup.event_control              cpuset.cpu_exclusive   cpuset.sched_load_balance
blkio.io_service_time_recursive    blkio.throttle.read_bps_device    cgroup.procs                      cpuset.cpus            cpuset.sched_relax_domain_level
```

图A-4　CGroup文件系统

因为CGroup并不涉及真实设备，所以mount的`dev name`参数可以自己随意命名，`mount dir`指的是具体挂载到哪个目录下的。命令执行完毕后，会在指定的挂载目录下出现很多文件，这些文件都是CGroup框架及各个子系统实现所生成的文件，我们后续就是通过操作这些文件来进行资源分配的。图A-5给出了CGroup文件系统挂载之后出现的一些文件，我们按照子系统将它们进行了分类。

如图A-5所示，CGroup按功能的不同分为不同的多组文件，图中深色背景对应的文件是单纯用于统计数据的只读文件。另外，CGroup框架本身提供的notify_on_release、release_agent及tasks文件是CGroup发展早期就存在的文件，将来有可能会被移除。

block子系统CFQ策略文件
blkio.weight_device
blkio.weight
blkio.leaf_weight_device
blkio.leaf_weight
blkio.time[_recursive]
blkio.sectors[_recursive]
blkio.io_merged[_recursive]
blkio.io_queued[_recursive]
blkio.io_wait_time[_recursive]
blkio.io_serviced[_recursive]
blkio.io_service_time[_recursive]
blkio.io_service_bytes[_recursive]

cpu子系统相关文件
cpu.shares
cpu.cfs_quota_us
cpu.cfs_period_us
cpu.rt_runtime_us
cpu.rt_period_us
cpu.stat

block子系统throttle策略文件
blkio.throttle.read_bps_device
blkio.throttle.write_bps_device
blkio.throttle.read_iops_device
blkio.throttle.write_iops_device
blkio.throttle.io_service_bytes
blkio.throttle.io_serviced

block子系统框架产生文件
blkio.reset_stats

cpu accounting子系统相关文件
cpuacct.usage
cpuacct.stat
cpuacct.usage_percpu

security子系统文件
devices.allow
devices.deny
devices.list

cpuset子系统相关文件
cpuset.cpu_exclusive
cpuset.cpus
cpuset.mem_exclusive
cpuset.mem_hardwall
cpuset.memory_migrate
cpuset.memory_pressure
cpuset.memory_pressure_enabled
cpuset.memory_spread_page
cpuset.memory_spread_slab
cpuset.mems
cpuset.sched_load_balance
cpuset.sched_relax_domain_level

CGroup框架相关文件
cgroup.clone_children
cgroup.event_control
cgroup.procs
notify_on_release
release_agent
tasks
cgroup.sane_behavior

memory子系统相关文件
memory.failcnt
memory.force_empty
memory.limit_in_bytes
memory.max_usage_in_bytes
memory.move_charge_at_immigrate
memory.numa_stat
memory.oom_control
memory.pressure_level
memory.soft_limit_in_bytes
memory.use_hierarchy
memory.swappiness
memory.usage_in_bytes
memory.stat

hugetlb相关文件
hugetlb.2MB.failcnt
hugetlb.2MB.limit_in_bytes
hugetlb.2MB.max_usage_in_bytes
hugetlb.2MB.usage_in_bytes

图A-5　CGroup各个子系统相关的文件

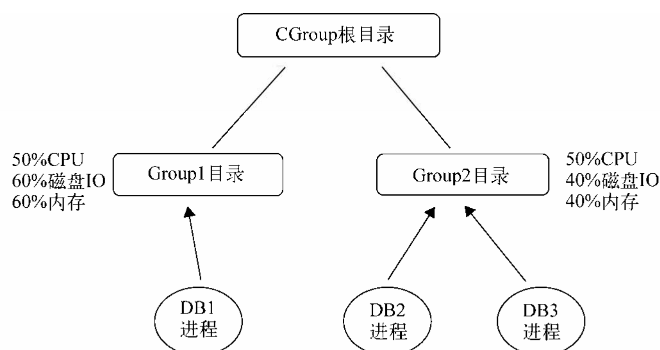
通常情况下，我们会在CGroup根目录中按照某种资源划分方式创建不同的目录，每个新创建的目录中也会自动生成相同的这些文件，如图A-6所示。

```
root@test:/sys/fs/cgroup# mkdir c1
root@test:/sys/fs/cgroup# ls c1/
blkio.io_merged                   blkio.leaf_weight                  blkio.time_recursive   cpu.rt_runtime_us                cpu.shares
blkio.io_merged_recursive         blkio.leaf_weight_device           blkio.weight           cpuset.cpu_exclusive             cpu.stat
blkio.io_queued                   blkio.reset_stats                  blkio.weight_device    cpuset.cpus                      devices.allow
blkio.io_queued_recursive         blkio.sectors                      cgroup.clone_children  cpuset.mem_exclusive             devices.deny
blkio.io_service_bytes            blkio.sectors_recursive            cgroup.event_control   cpuset.mem_hardwall              devices.list
blkio.io_service_bytes_recursive  blkio.throttle.io_service_bytes    cgroup.procs           cpuset.memory_migrate            freezer.parent_freezing
blkio.io_serviced                 blkio.throttle.io_serviced         cpuacct.stat           cpuset.memory_pressure           freezer.self_freezing
blkio.io_serviced_recursive       blkio.throttle.read_bps_device     cpuacct.usage          cpuset.memory_spread_page        freezer.state
blkio.io_service_time             blkio.throttle.read_iops_device    cpuacct.usage_percpu   cpuset.memory_spread_slab        notify_on_release
blkio.io_service_time_recursive   blkio.throttle.write_bps_device    cpu.cfs_period_us      cpuset.mems                      tasks
blkio.io_wait_time                blkio.throttle.write_iops_device   cpu.cfs_quota_us       cpuset.sched_load_balance
blkio.io_wait_time_recursive      blkio.time                         cpu.rt_period_us       cpuset.sched_relax_domain_level
```

图A-6　在CGroup文件系统中创建目录

实际上，CGroup内部实现本身就是通过维护一套树形结构来组织各个资源组的层次关系，而对外正好通过文件系统的目录树结构来表示出来，这样我们可以按照业务需求首先在CGroup文件系统上建立好对应的层次关系，然后再告诉CGroup系统哪些进程属于哪个资源组，从而完成对不同进程进行资源限制等工作。

例如，整个系统上同时混跑了3个数据库实例DB1、DB2和DB3，假设不同实例的访问量是不一样的，我们要根据实际访问情况为不同的实例分配不同的资源占比，例如希望DB1独占50%的CPU资源、60%的磁盘IO资源以及60%的内存资源，而其余两个实例共享剩余的资源，那么就可以在CGroup文件系统上创建两个目录，分别代表两个资源组，然后分别配置这两个资源组的资源配额，最后将这些实例加入到不同的资源组中来完成上面所述的目标，如图A-7所示。



图A-7　CGroup资源控制

每个资源组资源额度的配置是通过将相关内容写入到各子系统自动生成的文件中来完成的，后续我们会详细说明。现在我们知道了通过CGroup文件系统来组织资源的层次关系，那么如何告诉CGroup哪些进程属于哪些资源组呢？这可以通过修改cgroup.tasks或者cgroup.procs这两个文件来完成，即将待加入进程的进程ID写入这两个文件中的一个。命令如下：

```
root# echo PID > 资源组相应目录/tasks
```

或者

```
root# echo TGID > 资源组相应目录/procs
```

那么，这两个文件有什么区别呢，下面来解释一下。

- tasks文件：写入这个文件的数值被认为是进程的PID。将某个线程的PID写入这个文件，那么这个线程本身将被加入该资源组进行管理。
- procs文件：写入这个文件的数值被认为是进程的TGID，即线程组ID。将一个进程的TGID写入该文件，那么属于该进程的其他线程将自动加入该资源组进行管理。

总体上，CGroup的基本使用就是这样一个过程：按照进程资源的种类划分为不同的资源组，

在CGroup文件系统上建立相应的目录结构关系，配置相应目录中各个子系统的资源配额文件，最后将不同的进程加入到不同的资源组中，即写入进程的PID或者TGID到相应目录中的tasks或procs文件中就可以了。接下来，我们会结合一些实际的案例来分别详细讲述各个子系统的配置。

A.2.3　使用CGroup限制进程CPU使用带宽

首先，我们来看一个限制CPU使用带宽的例子。CGroup通过cpu.cfs_quota_us和cpu.cfs_period_us这两个文件来进行资源组的CPU带宽限制，这两个文件的默认值是：

```
root@test:/sys/fs/cgroup# cat cpu.cfs_quota_us
-1
root@test:/sys/fs/cgroup# cat cpu.cfs_period_us
100000
```

实际代表的含义是在cpu.cfs_period_us指定的单位内（微秒）能够使用cpu.cfs_quota_us指定的CPU时间（微秒）。cpu.cfs_quota_us的默认值为−1，代表没有限制。

下面我们简单写一段代码来说明这两个文件的作用：

```
// cpu_test.c

#define _GNU_SOURCE
#include <sched.h>
int main()
{
    cpu_set_t set;

    CPU_ZERO(&set);
    CPU_SET(0, &set);
    sched_setaffinity(0, sizeof(cpu_set_t), &set);

    while(1);

    return 0;
}
```

这段代码的逻辑非常简单，首先通过sched_setaffinity来绑定当前进程运行在0号CPU，然后就是直接一个死循环。之所以要绑定CPU，主要是避免多CPU带来的影响。

直接编译运行该程序：

```
root@test:~# gcc -o cpu_test cpu_test.c
root@test:~# ./cpu_test
...程序无限循环...
```

然后开启另一个窗口运行top：

```
...
PID     USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM  TIME+    COMMAND
12410   root      20  0   4156  352  276  R  100   0.0   3:38.27  cpu_test
...
```

可以看到，默认情况下这个死循环的程序占用了100%的CPU带宽，下面我们创建一个资源组并加以限制：

```
root@test:/sys/fs/cgroup# mkdir group1
root@test:/sys/fs/cgroup# cd group1

root@test:/sys/fs/cgroup/group1# echo "0-3" > cpuset.cpus
root@test:/sys/fs/cgroup/group1# echo 0 > cpuset.mems

root@test:/sys/fs/cgroup/group1# cat cpu.cfs_period_us
100000
root@test:/sys/fs/cgroup/group1# echo 50000 > cpu.cfs_quota_us
root@test:/sys/fs/cgroup/group1# echo 12410 > tasks
```

然后再执行top命令：

```
PID    USER      PR  NI  VIRT    RES  SHR    S %CPU    %MEM    TIME+      COMMAND
12410  root      20  0   4156    352  276    R 50      0.0     13:06.80   cpu_test
```

可以看到，同一个进程没有重新运行，CPU的带宽直接就变成50%，这是因为我们设置了在100 000（cpu.cfs_period_us）这个时间单位内允许该资源组使用50 000（cpu.cfs_quota_us）这么多时间单位的CPU。

重新设置资源配额，再次验证下：

```
root@test:/sys/fs/cgroup/group1# echo 25000 > cpu.cfs_quota_us
```

继续执行top命令：

```
PID    USER   PR  NI  VIRT  RES  SHR  S   %CPU      %MEM      TIME+      COMMAND
12410  root   20  0   4156  352  276  R   25        0.0       15:15.68   cpu_test
```

可以看到我们对CPU资源限额的效果。

这里再简单说明一下，上面的命令操作了cpuset.cpus和cpuset.mems这两个文件。因为新创建的资源组必须要设置这两个文件的值，否则无法将进程加入资源组。这两个文件分别代表资源组允许使用的CPU以及内存节点。我的测试环境有4个CPU，所以我写入了0~3到cpuset.cpus文件，表示该资源组的进程可以使用任意一个CPU，同时由于我的机器是UMA（均匀存储器存取）的机器，只有一个内存节点，所以将0写入了cpuset.mems这个文件。

下面再简单介绍一下CGroup里面CPU子系统中其他几个文件的作用。

- cpu.shares：主要用于CPU的权重控制。我们可以为不同的资源组配置不同的权重，但是由于这个权重值实际代表的是一个资源组使用CPU的下限值，多个资源组在运行时会按照权重分配，如果其中资源组的一个进程暂时不会占用CPU，例如正在等待某种资源，那么其他资源组的进程会来抢占CPU资源，所以这个值本身不是准确的可以衡量的值，所以在通常情况下我们没有使用cpu.shares。
- cpu.rt_period_us和cpu.rt_runtime.us：与之前的cfs_period_us和cfs_quota_us这两个文件在功能上类似，不同之处在于以rt开头的这两个文件是作用于实时进程的，而以cfs开头的两个文件则作用于普通进程。

我们可以通过sched_setscheduler系统调用将进程设置为实时进程，这里不再详细展开，有兴趣的读者可以自行试验。

A.2.4 使用CGroup限制进程的内存使用

CGroup对内存的限制功能可以通过memory.limit_in_bytes这个文件完成，这里我们先看一段程序：

```
// mem_test.c

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

#define MALLOC_SIZE 1024 * 1024

int main(int argc, char **argv)
{
    char *s;

    setbuf(stdout, NULL);

    while (1) {
        s = malloc(MALLOC_SIZE);
        printf(".");

        .memset(s, 0x1, MALLOC_SIZE);

        sleep(1);
    }
}
```

这段程序的功能很简单，通过循环不断分配内存，每次分配1MB，随后马上通过memset来写这块内存以引发缺页中断，其目的是为了分配对应的物理内存，程序每分配1MB睡眠1秒。

编译运行该程序：

```
root@test:~# gcc -o mem_test mem_test.c
root@test:~# ./mem_test
... 无限循环中
```

开启一个新的终端执行命令：

```
root@test:~# top -p `pgrep mem_test`
...
PID     USER      PR  NI  VIRT     RES    SHR  S    %CPU   %MEM    TIME+     COMMAND
17948   root      20  0   73852    68m    344  S    0      1.9     0:00.07    mem_test
...
```

此时程序会每隔1秒钟增长1MB的物理内存占用，这可以通过top的RES值看到。

现在我们希望将程序使用的物理内存控制在5MB以内，可以进行如下操作：

```
root@test:~# cd /sys/fs/cgroup/
root@test:/sys/fs/cgroup# mkdir memcgroup
root@test:/sys/fs/cgroup# cd memcgroup/

root@test:/sys/fs/cgroup/memcgroup# echo "5M" > memory.limit_in_bytes
root@test:/sys/fs/cgroup/memcgroup# echo "0-3" > cpuset.cpus
root@test:/sys/fs/cgroup/memcgroup# echo 0 > cpuset.mems
root@test:~# ./mem_test
... 无限循环中
```

开启新终端：

```
root@test:/sys/fs/cgroup/memcgroup# echo `pgrep mem_test` > tasks
root@test:/sys/fs/cgroup/memcgroup# top -p `pgrep mem_test`
PID     USER     PR  NI  VIRT   RES   SHR S  %CPU   %MEM     TIME+     COMMAND
18135   root     20  0   40588  4592  344 S   0      0.1      0:00.09   mem_test
```

可以看到，内存占用被限制在了5MB以内。

重新调整这个参数：

```
root@test:/sys/fs/cgroup/memcgroup# echo "10M" > memory.limit_in_bytes
root@test:/sys/fs/cgroup/memcgroup# top -p `pgrep mem_test`
PID     USER     PR  NI  VIRT   RES  SHR  S %CPU  %MEM    TIME+      COMMAND
18135   root     20  0   40588  10m  344  S  0    0.1      0:00.09   mem_test
```

可以看到调整后的效果，RES值飙升到了10MB。

当程序使用的内存超出memory.limit_in_bytes时，系统会根据另外一个文件memory.memsw.limit_in_bytes的值来确定是否将超出的内存换出。实际上，memory.memsw.limit_in_bytes的值代表可用内存加上可以换出的大小，即包含了memory.limit_in_bytes的值。我们看一下memory.

memsw.limit_in_bytes文件的默认值：

```
root@test:/sys/fs/cgroup/memcgroup# cat memory.memsw.limit_in_bytes
18446744073709551615
```

可以看到，默认值是一个非常大的值，所以我们上面的程序在分配了超过memory.limit_in_bytes的内存后，就开始发生换出操作，即一部分内存页被换到了磁盘上，通过vmstat可以看到这一点。重新运行上面的程序，然后运行vmstat：

```
root@test:/sys/fs/cgroup/memcgroup# vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b    swpd   free      buff      cache      si    so     bi    bo    in    cs us sy id wa
...
 0  0    5604 1443976 137136 865588    0     0     0    44  153   337   1   0 99  1
 0  0    5604 1443256 137136 865412    0     0     0     0  130   268   1   0 99  0
 0  0   10852 1445604 137136 865452    0  5248     0  5424  194   320   1   1 98  0
 0  0   10852 1444868 137136 865636    0     0     0     0  142   310   1   0 99  0
 0  0   10852 1443604 137136 865412    0     0     0     0  134   286   0   0 99  0
 1  0   13924 1444564 137136 865560    0  3072     0  3072  141   260   1   0 99  0
 0  0   13924 1443852 137144 865456    0     0     0    24  163   356   1   1 98  1
 0  0   13924 1443488 137144 865336    0     0     0     0  153   434   1   0 99  0
 1  0   16100 1445016 137144 865560    0  2176     0  2176  208   706   1   1 98  0
 ...
```

在程序运行了10秒钟之后，我们通过vmstat命令看到swap的so列出现了页换出的现象。

如果我们不希望程序在超过内存限额的情况下去使用交换区，那么可以将memory.memsw.limit_in_bytes文件与memory.limit_in_bytes设置相同的值：

```
root@test:/sys/fs/cgroup/memcgroup# cat memory.limit_in_bytes > memory.memsw.limit_in_bytes
root@test:/sys/fs/cgroup/memcgroup# cat memory.limit_in_bytes
10485760
root@test:/sys/fs/cgroup/memcgroup# cat memory.memsw.limit_in_bytes
10485760
```

重新运行测试程序：

```
root@test:~# ./mem_test
...循环中
```

开启新终端：

```
root@test:/sys/fs/cgroup/memcgroup# echo `pgrep mem_test` > tasks
```

运行10秒钟之后，会看到：

```
root@test:~# ./mem_test
...Killed
```

我们的程序被杀死了。运行dmesg验证一下：

```
[22866.207485] [ pid ]   uid  tgid total_vm      rss nr_ptes swapents oom_score_adj name
[22866.207549] [ 6832]     0  6832     4123     3188      13        0             0 mem_test
[22866.207554] Memory cgroup out of memory: Kill process 6832 (mem_test) score 1226 or sacrifice child
[22866.207560] Killed process 6832 (mem_test) total-vm:16492kB, anon-rss:12376kB, file-rss:376kB
```

可以看到，我们的程序确实被操作系统杀死了。假如希望程序在使用内存超过限额时不被马上杀死，而是等待一段时间，并且如果这段时间内我们重新调整了内存限额策略，或者程序自己释放了一些内存从而不再超过限额，程序可以自己恢复继续执行，这个时候可以通过操作memory.oom_control文件来完成。首先看看这个文件的内容：

```
root@test:/sys/fs/cgroup/memcgroup# cat memory.oom_control
oom_kill_disable 0
under_oom 0
```

这个文件同时提供了两个值，一个用来写入，一个用来查看。我们可以通过向memory.oom_control写入1来改变`oom_kill_disable`的值，其默认值是0，即不会禁止kill操作，这里我们把它改成1（表示当内存超过限额时不杀死进程）：

```
root@test:/sys/fs/cgroup/memcgroup# echo 1 > memory.oom_control
root@test:/sys/fs/cgroup/memcgroup# cat memory.oom_control
oom_kill_disable 1
under_oom 0
```

然后重新运行mem_test程序：

```
root@test:~# ./mem_test
...程序循环
```

开启新终端：

```
root@test:/sys/fs/cgroup/memcgroup# echo `pgrep mem_test` > tasks
```

10秒钟之后，我们看到：

```
root@test:~# ./mem_test
...程序无响应（程序正常运行会不断打印.）
```

查看oom_control文件：

```
root@test:/sys/fs/cgroup/memcgroup# cat memory.oom_control
oom_kill_disable 1
under_oom 1
```

我们看到这时`under_oom`为1，它用来告诉我们当前资源组是否处于`oom`状态，这时由于我们关闭了该资源组内存达到限制时被杀死的功能，所以程序表现为无响应状态。下面通过top命令来查看进程的状态：

```
root@test:/sys/fs/cgroup/memcgroup# top -p `pgrep mem_test`
...
PID  USER   PR  NI  VIRT   RES  SHR  S  %CPU  %MEM    TIME+      COMMAND
6947 root   20  0   16492  12m  376  D  0     0.4     0:00.01    mem_test
...
```

通过top可以看到程序处于D状态，即不可打断的睡眠状态，这时重新调整memory限额策略：

```
root@test:/sys/fs/cgroup/memcgroup# echo "20M" > memory.memsw.limit_in_bytes
root@test:/sys/fs/cgroup/memcgroup# echo "20M" > memory.limit_in_bytes
```

可以看到测试程序开始继续运行：

```
root@test:~# ./mem_test
...程序继续运行
```

新建终端并执行top命令来查看进程的最新状态：

```
root@test:/sys/fs/cgroup/memcgroup# top -p `pgrep mem_test`
...
 PID USER   PR  NI  VIRT   RES  SHR  S  %CPU   %MEM    TIME+       COMMAND
 6947root   20  0   16492  12m  376  R  0      0.4     0:00.01     mem_test
...
```

可以看到程序重新变成R状态，即运行状态（这里也很可能是S状态，因为申请1MB内存是非常快的，程序很快进入睡眠状态，即S状态）。

除了可以简单控制资源组内存限额中的oom（out of memory）机制外，CGroup同时还为我们提供了方便的通知机制，例如在某个资源组发生了oom事件时可以给我们发送通知。通过操作cgroup.event_control文件，可以完成事件通知的注册工作，具体过程如下。

(1) 创建一个eventfd。

(2) 打开一个感兴趣并可以用作通知机制的文件，例如memory.oom_control。

(3) 向cgroup.event_control文件中写入如下格式的字符串：<event fd> <fd of memory.oom_conrol>。

CGroup对应用暴露的这个通知接口的使用方式看起来可能比较陌生，下面我们通过一段代码来了解一下：

```
// mem_event.c

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/eventfd.h>
#include <sys/epoll.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
```

```
#include <stdlib.h>

#define MAX_EVENTS 10
#define BUF_SIZE 512

void usage()
{
    printf("usage: mem_event <cgroup.event_control> <memory.oom_control>\n");
    exit(EXIT_FAILURE);
}

void error_exit(char *err_msg)
{
    printf("%s,error: %s(%d)\n", err_msg, strerror(errno), errno);
    exit(EXIT_FAILURE);
}

int main(int argc, char **argv)
{
    uint64_t u;
    char buf[BUF_SIZE];
    int efd, ctl_fd, oom_fd, n, ep_fd, nfds;
    struct epoll_event ev, events[MAX_EVENTS];

    if (argc != 3) {
        usage();
    }

    if ((efd = eventfd(0, EFD_NONBLOCK)) == -1) {
        error_exit("create eventfd failed");
    }

    if ((ctl_fd = open(argv[1], O_WRONLY)) == -1) {
        error_exit("open cgroup.event_control failed");
    }

    if ((oom_fd = open(argv[2], O_RDONLY)) == -1) {
        error_exit("open cgroup.oom_control failed");
    }

    if ((n = snprintf(buf, BUF_SIZE, "%d %d", efd, oom_fd)) >= BUF_SIZE) {
        error_exit("event buf truncated");
    }

    if (write(ctl_fd, buf, n) == -1) {
        error_exit("write cgroup.event_control failed");
    }

    if (close(ctl_fd) == -1) {
```

```
        error_exit("close cgruop.event_control failed");
    }

    if ((ep_fd = epoll_create(1024)) == -1) {
        error_exit("epoll create failed");
    }

    ev.events = EPOLLIN;
    ev.data.fd = efd;

    if (epoll_ctl(ep_fd, EPOLL_CTL_ADD, efd, &ev) == -1) {

        error_exit("add eventfd to epoll failed");
    }

    for (;;) {
        nfds = epoll_wait(ep_fd, events, MAX_EVENTS, -1);

        if (nfds == -1) {
            error_exit("epoll_wait error");
        }

        for (n = 0; n < nfds; n++) {
             if (events[n].data.fd != efd) {
                 error_exit("epoll receive unknown event");
             }

             if (read(efd, &u, sizeof(uint64_t)) != sizeof(uint64_t)) {
                 error_exit("read from eventfd error");
             }

             printf("oom event received\n");
        }
    }
}
```

编译运行该程序：

```
root@test:~# gcc -o mem_event mem_event.c
root@test:~# ./mem_event /sys/fs/cgroup/memcgroup/cgroup.event_control
/sys/fs/cgroup/memcgroup/memory.oom_control
<程序无限等待>(epoll无限等待来自CGroup的通知)
```

新建终端重新运行mem_test：

```
root@test:~# ./mem_test
...<程序运行中>
```

新建另一个终端并执行以下命令：

```
root@test:/sys/fs/cgroup/memcgroup# echo `pgrep mem_test` > tasks
```

10秒钟之后可以看到：

```
root@test:~# ./mem_event /sys/fs/cgroup/memcgroup/cgroup.event_control
/sys/fs/cgroup/memcgroup/memory.oom_control
oom event received
```

epoll监听的终端收到了oom的消息。由于之前设置了oom_control中的oom_kill_disable为1，所以我们的mem_test程序并没有被杀掉：

```
root@test:~# ./mem_test
...<无限等待中>
```

假如我们希望在资源紧张的情况下提前获得通知，而不是等到已经oom的时候，我们还可以注册另外一个用作通知的文件memory.pressure_level。

该文件可以将内存使用情况按级别分类，分别是low、medium、critical三个级别。用户可以指定级别，在内存使用情况超过用户设置的级别时就通知用户。下面分别解释下这三个级别的不同含义。

- **low级别**：当系统开始为新的内存申请而回收内存时，处于此级别。
- **medium级别**：在此级别时，系统内存使用已经处于中度紧张中，系统可能已经使用交换区。
- **critical级别**：在此级别时，系统内存处于严重紧张中，系统内存的换入换出会比较严重。

我们可以按如下流程操作memory.pressure_level文件，来进行内存资源使用情况的通知注册。

(1) 创建eventfd句柄。
(2) 打开 memory.pressure_leven文件。
(3) 向cgroup.event_control文件中写入如下格式的字符串：

```
<event_fd> <fd of memory.pressure_level> <level>
```

其中level可以是前面描述的low、medium、critical。

我们可以简单修改一下mem_event.c的代码，来测试pressure_level的通知功能。这里我们只需要简单修改下面几行代码。

原代码：

```
if ((n = snprintf(buf, BUF_SIZE, "%d %d", efd, oom_fd)) >= BUF_SIZE) {
    error_exit("event buf truncated");
}
```

修改为：

```
if ((n = snprintf(buf, BUF_SIZE, "%d %d low", efd, oom_fd)) >= BUF_SIZE) {
    error_exit("event buf truncated");
}
```

然后执行程序，将第二个命令行参数改为pressure_level文件路径即可：

```
root@test:~# ./mem_event /sys/fs/cgroup/memcgroup/cgroup.event_control
/sys/fs/cgroup/memcgroup/memory.pressure_level
```

程序运行几秒后即会收到通知。若读者有兴趣，可以自行将low改为medium或者critical进行详细测试，这里不再一一列出。

A.2.5　使用CGroup限制磁盘带宽使用

磁盘IO带宽的控制可以通过操作CGroup的blkio.weight文件及blkio.weight_device文件来完成，具体操作格式如下：

```
root# echo <10-1000> > blkio.weight
root# echo <major>:<minor> <10-1000> > blkio.weight_device
```

blkio.weight接受一个10~1000的值，代表该资源组磁盘IO的一个权重，而实际IO带宽的分配是根据此权重值来完成的。

blkio.weight_device与blkio.weight的含义基本相同，区别在于blkio.weight-device可以单独限制某个设备的权重值，而blkio.weight则代表所有设备。

这样我们可以在设置了blkio.weight的同时单独为某个设备重新分配不同的权重值，即覆盖掉默认的设置。

这里需要特殊说明的是，blkio.weight 磁盘IO带宽限制的实现是通过内核通用块层的CFQ调度器来完成的，所以我们使用这种方式进行IO控制时，必须首先确认使用的是CFQ IO调度器。这可以通过下面的方式进行设置。

首先，确认我们测试所在的文件系统挂载点所属的盘符。

```
root@test:~# lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda         8:0           0     465.8G    0  disk
 ├─sda1      8:1           0     462.2G    0  part      /
 ├─sda2      8:2           0     1K        0  part
 └─sda5      8:5           0     3.6G      0  part [SWAP]
sr0         11:0          1     1024M     0 rom
```

从上面的命令可以看到我们的盘符是sda。接下来我们看下这块盘的IO调度器是什么，这可以通过执行如下命令来查看：

```
root@test:~# cat /sys/block/sda/queue/scheduler
noop [deadline] cfq
```

可以看到，当前默认的IO调度器是deadline，通过如下命令将其更改为CFQ：

```
root@test:~# echo "cfq" > /sys/block/sda/queue/scheduler
root@test:~# cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
```

此时就可以测试磁盘IO带宽控制的效果了。

下面我们来看一个例子。新建两个不同的资源组，分别分配不同的权重值：

```
root@test:/sys/fs/cgroup# mkdir blktest1
root@test:/sys/fs/cgroup# mkdir blktest2
root@test:/sys/fs/cgroup# echo 900 > blktest1/blkio.weight
root@test:/sys/fs/cgroup# echo 100 > blktest2/blkio.weight
```

然后分别开启两个终端并执行以下命令。

终端1：

```
root@test:~# echo $$ > /sys/fs/cgroup/blktest1/tasks
```

终端2：

```
root@test:~# echo $$ > /sys/fs/cgroup/blktest2/tasks
```

这里解释一下，在shell下用$$代表当前shell进程的PID。这两条命令分别把当前shell进程加入到相应资源组中。由于CGroup的实现机制是子进程会继承父进程的资源组配置，所以我们后续在这两个终端中运行的所有命令都会属于相应的资源组。

随后，我们分别在两个终端执行dd命令进行测试。

终端1：

```
root@test:~# dd if=/dev/zero of=./test1.data oflag=direct bs=1M count=10240
```

终端2：

```
root@test:~# dd if=/dev/zero of=./test2.data oflag=direct bs=1M count=10240
```

终端3：

```
root@test:~# iotop
Total DISK READ:      0.00 B/s | Total DISK WRITE:     127.91 M/s
TID    PRIO    USER   DISK READ  DISK WRITE  SWAPIN     IO>              COMMAND
16149  be/4    root   0.00 B/s   116.37 M/s  0.00 %     97.01%   dd if=/dev/zero of=./test1.data
                                                                  oflag=direct bs=1M count=10240
```

```
16151  be/4   root   0.00 B/s   11.54 M/s   0.00 %    92.01%  dd if=/dev/zero of=./test2.data
                                                               oflag=direct bs=1M count=10240
...
```

通过iotop命令看到DISK WRITE一栏上两个进程的写入能力大约为9：1。这里特别说明一下，当前内核版本（3.13）只能支持直接IO（direct IO）方式的磁盘IO资源控制，其主要原因是内核的资源控制实现是按照进程来划分的，这也是我们要把待控制的进程PID写入tasks文件的原因。但是对于缓存IO（buffer IO）的写入，由于它只是写入到内核的页面缓存中，需要等到将来某一时刻会异步刷新到磁盘上，所以真正写磁盘时，进程可能根本已经不是当初发起写的那个进程，所以基于缓存IO方式的磁盘IO是无法准确控制资源限额的。所以我们在用dd命令进行测试时，需要加上oflag=direct参数，这样实际的写入就是直接IO方式的了。

基于带宽控制方式的好处是整体上的控制策略是弹性的。例如，虽然我们限制了两个资源组的权重，但是假如其中一个资源组没有活动的进程或者使用的IO资源并不多，那么另一个进程是可以使用剩余的IO资源的。我们进行如下测试进行验证。

终端1：

```
root@test:/sys/fs/cgroup# cat blktest1/blkio.weight
900
root@test:/sys/fs/cgroup# cat blktest2/blkio.weight
100
```

终端2：

```
root@test:~# echo $$ > /sys/fs/cgroup/blktest2/tasks
root@test:~# dd if=/dev/zero of=test2.data oflag=direct bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 7.99986 s, 134 MB/s
```

同前面的测试对比一下可以看出差别。在前面的测试案例中，两个dd程序一起运行时，由于blktest2资源组配置的带宽权重为100，所以最终的写入能力在10MB/s左右。而在第二个测试中，由于只运行了一个dd程序，虽然其所在资源组的权重配额仍然是100，但是由于没有其他进程在使用IO资源，所以可以占用全部的IO资源，这里体现了我们所讨论的弹性。

A.2.6　通过限流方式控制IO资源的使用

虽然基于IO带宽权重的资源控制方式能够带来弹性，但是它在很多场景下都是无法工作的，例如我们的数据库应用通常会使用限期（deadline）IO调度器，因为该调度器内部会单独维护一个队列，IO请求在队列中按请求时间的先后顺序进行排列，所以该调度器在单个IO请求延迟方面是有严格保障的，所以对于在线的数据库应用来说，整体系统表现就会好于CFQ IO调度器，并且在很多高速设备上CFQ IO调度器的表现也不如限期IO调度器理想。

除了调度器的因素外，很多高速设备（例如Fusion-IO卡的驱动）或者虚拟设备（例如dm io）根本不走IO调度器，所以也就没有办法使用依赖CFQ IO调度器的IO带宽控制策略，这时我们需要一种更通用的IO控制方法，那就是基于限流的IO控制策略。该策略的配置主要通过操作图A-8中的几个文件来完成。

blkio.throttle.read_bps_device
blkio.throttle.write_bps_device
blkio.throttle.read_iops_device
blkio.throttle.write_iops_device
blkio.throttle.io_service_bytes
blkio.throttle.io_serviced

图A-8 block子系统throttle策略文件

下面简单解释一下各个文件的含义。

- blkio.throttle.read[write]_bps_device文件：这两个文件分别用来限制资源组针对某个设备每秒钟允许的读写字节数，即属于该资源组的所有进程加一起每秒钟只能读blkio.throttle.read_bps_device字节的数据以及写blkio.throttle.write_bps_device字节的数据到指定设备。

  该文件写入的格式为：

  <主设备号>:<次设备号> <允许读取或写入的字节数>

- blkio.throttle.read[write]_iops_device文件：这两个文件分别用来限制资源组针对某个设备每秒钟允许的读写次数。

  该文件写入的格式为：

  <主设备号>:<次设备号> <允许读取或写入的次数>

- blkio.throttle.io_serviced和blkio.throttle.io_service_bytes文件：这两个文件为只读文件，用于统计该资源组当前流过的字节数及IO次数。

下面我们还是简单看个测试例子。

首先，我们通过lsblk命令来查看测试文件系统对应的磁盘设备号：

```
root@test:~# lsblk
NAME   MAJ:MIN RM   SIZE     RO TYPE MOUNTPOINT
```

```
sda          8:0         0     465.8G    0    disk
 ├─sda1      8:1         0     462.2G    0    part        /
 ├─sda2      8:2         0     1K        0    part
 └─sda5      8:5         0     3.6G      0    part [SWAP]
sr0          11:0        1     1024M     0    rom

root@test:/sys/fs/cgroup# echo "8:0 1048576" > blktest1/blkio.throttle.write_bps_device
```

这里我们限制blktest1资源组每秒钟只能写入1MB数据到sda设备。将当前shell加入资源组并用dd程序验证一下：

```
root@test:~# echo $$ > /sys/fs/cgroup/blktest1/tasks
root@test:~# dd if=/dev/zero of=test1.data oflag=direct bs=1M count=8
8+0 records in
8+0 records out
8388608 bytes (8.4 MB) copied, 8.00016 s, 1.0 MB/s
```

可以看到，在没有其他资源组的程序竞争的情况下，blktest1资源组对应的进程被严格限制在了1MB/s的写入能力，这里也验证了我们所提到的基于限流的IO控制方式缺少弹性的问题。

我们再测试一下在同一资源组内同时运行两个dd程序的情况。

终端1：

```
root@test:~# echo $$ > /sys/fs/cgroup/blktest1/tasks
root@test:~# dd if=/dev/zero of=test1.data oflag=direct bs=1M count=8
8+0 records in
8+0 records out
8388608 bytes (8.4 MB) copied, 15 s, 559 kB/s
```

终端2：

```
root@test:~# echo $$ > /sys/fs/cgroup/blktest1/tasks
root@test:~# dd if=/dev/zero of=test1.data oflag=direct bs=1M count=8
8+0 records in
8+0 records out
8388608 bytes (8.4 MB) copied, 14.0023 s, 599 kB/s
```

可以看到，两个进程加在一起每秒钟写入了大约1MB的数据，这是符合预期的。

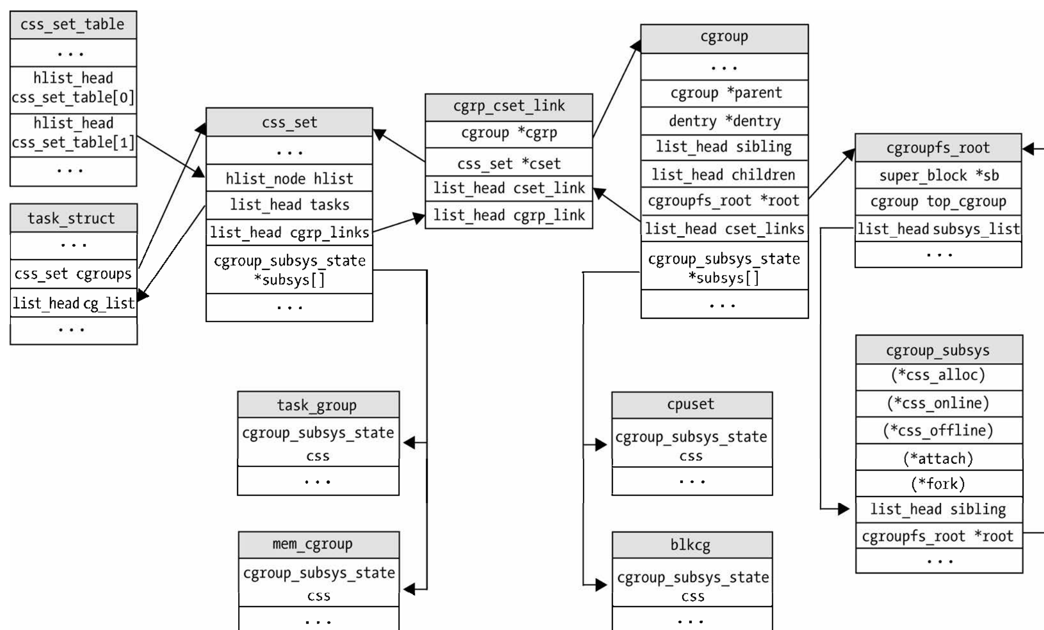
A.3　CGroup框架实现

前面我们大致熟悉了CGroup的使用，下面我们来简单分析一下CGroup的实现。首先，我们需要清楚CGroup本身只是一个通用的资源管理框架，各个子系统遵循此框架来实现自己子系统的具体资源管理策略。这也是Linux内核惯用的一种机制与策略分离的设计方法。实际上，CGroup本身只是机制和框架，而各子系统都有自己的策略。下面我们首先来了解一下CGroup框架机制

的实现，然后再讨论其中一个子系统的实现。

A.3.1　核心数据结构

首先，我们给出了CGroup框架核心数据结构之间的一个关系图，如图A-9所示。



图A-9　CGroup核心数据结构

- ❑ **cgroup_subsys**：该数据结构定义了CGroup各子系统需要实现的接口，每个子系统都有自己的一个cgroup_subsys实例，例如mem_cgroup_subsys定义了内存子系统的实现，blkio_subsys定义了通用块子系统的实现。各子系统通过定义此数据结构的实例并实现对应的接口来提供各自独立的策略。
- ❑ **cgroupfs_root**：该数据结构代表的是我们挂载的CGroup文件系统的根目录，其中sb指向CGroup文件系统超级块的实例，top_cgroup代表了资源组的根节点，subsys_list是一个包含cgroup_subsys实例的链表。这里解释一下，如果我们挂载CGroup文件系统时没有指定任何参数，则系统会默认把所有的子系统一起挂载到指定挂载点上，即这时cgroupfs_root里包含了所有的cgroup_subsys的实例。此外，我们也可以在挂载的时候指定挂载哪个子系统，这样生成的cgroupfs_root就只包含指定的cgroup_subsys实例。例如，我们只希望挂载通用块子系统，则可以通过如下命令完成：

```
root# mount -t cgroup -o blkio none /sys/fs/cgroup
```

- cgroup：该数据结构是整个CGroup框架最核心的一个数据结构。实际上，在CGroup文件系统下创建的每个目录都对应一个cgroup数据结构，该数据结构通过parent、sibling、children这3个成员构成了一颗和文件系统目录结构完全一致的树型结构。cgroup的dentry成员的作用是与VFS结合在一起。cgroup_subsys_state数组保存了该资源组下对应的所有子系统的核心数据结构。从图A-9中可以看出，其关联的cpuset、blkcg等即为不同子系统的核心数据结构，其中都包含了cgroup_subsys_state这个成员。实际上，这里是内核常用的一种多态的设计方法，即cgroup数据结构里面的cgroup_subsys_state[]实际保存的就是cpuset、blkcg这些东西，将来需要通过内核的container_of宏来还原相应的数据结构，这在后面再详细讨论。

其实我们在使用CGroup时，所做的挂载文件系统、创建相应目录以指定资源组的关系以及对应的一些配置工作，通过上面的3个核心数据结构就已经可以完成了，这一部分相当于一个静态的资源组织框架。我们后面要做的是将某些进程加入到对应的资源组中，这个过程是动态的，其实简单地说就是如何让代表进程的task_struct数据结构与相应的cgroup数据结构关联起来。

一种最直接的实现方式可能就是直接在task_struct数据结构上加入对相应cgroup数据结构的引用，但是我们仔细考虑一下就知道这样是行不通的。实际上，进程与相应资源组并不是简单的一对一的关系，而是多对多的关系。首先，一个资源组允许多个进程加入，其次一个进程可能同时加入不同的资源组。例如，下面的案例就是这种情况：

```
root# mount -t cgroup -o blkio blkio_cg /sys/fs/cgroup/blkio
root# mount -t cgroup -o cpu cpu_cg /sys/fs/cgroup/cpu

root# echo $$ > /sys/fs/cgroup/blkio/tasks
root# echo $$ > /sys/fs/cgroup/cpu/tasks
```

由于我们把不同的子系统分别挂载到了不同的目录上，那么假如一个进程同时要限制IO访问和CPU带宽，就需要把这个进程同时加入这两个不同的资源组中，这实际上就是一个进程对应多个资源组的情况。这和在一个挂载的CGroup文件系统下将进程在不同资源组间移动是完全不同的，后面这种情况下，进程加入另一个资源组的同时就自动在之前的资源组移除掉了。

如图A-9所示，内核通过引入css_set数据结构和cgrp_cset_link来将task_struct与cgroup之间的多对多关系表现出来。我个人认为这样做简化了task_struct数据结构，因为task_struct已经非常庞大了，如果直接让task_struct与cgroup建立多对多的关系，则必然还有一些相关成员要记录在task_struct数据结构上面，而通过引入css_set数据结构来简化task_struct数据结构更加合理。

- cgrp_cset_link：该数据结构完全是为了管理css_set与cgroup之间多对多的关系而存在的。cgrp成员指向对应的cgroup，cset成员指向对应的css_set，之后以cgroup数据结构的

cset_links 为链表头串联了所有的 cgrp_cset_link 数据结构，这样通过 cgroup 的 cset_links链表可以遍历该资源组cgroup对应的所有css_set。由于css_set保存了所有的 task_struct，这样也就可以遍历cgroup资源组下面所有的进程task_struct。同理，css_set 里的cgrp_links链表头通过cgrp_cset_link保存了该css_set加入的所有资源组cgroup，也就是可以遍历出一个进程加入的所有资源组。

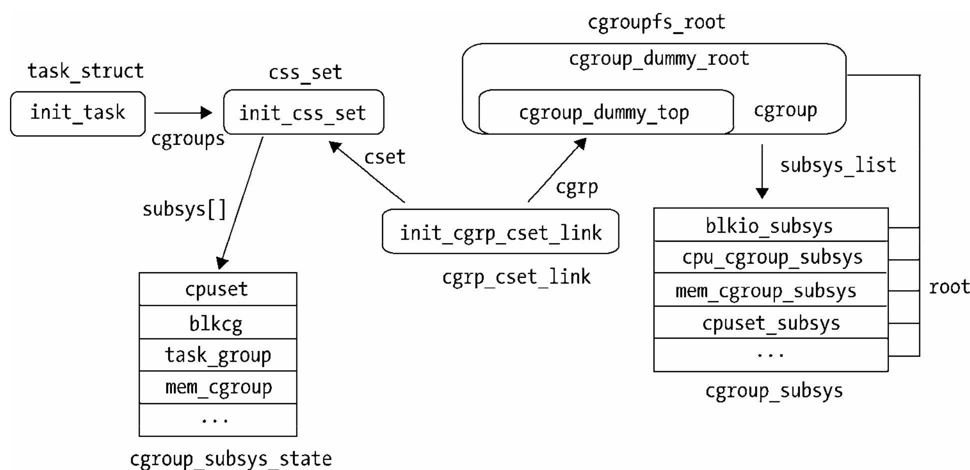
- **css_set**：该数据结构的tasks成员保存了task_struct的链表，cgrp_links链表头前面已经介绍过，subsys[]数组保存了该css_set加入的所有资源组对应的各子系统核心数据结构实例。
- **css_set_table**：整个系统有一个全局的哈希表css_set_table，hlist成员用来链接哈希表内同一个桶内的所有css_set实例。css_set_table哈希表的作用在于当我们需要建立task_struct与cgroup的关系时，需要确认当前系统是否已经存在一个可以复用的css_set数据结构，例如两个进程所加入的资源组是完全相同的，那么它们是可以共享css_set数据结构的。
- **task_struct**：该数据结构通过cgroups成员来访问css_set，随后通过css_set的subsys[]数组来得到各子系统在该资源组的核心数据结构的实例。我们对每个资源组的一些具体配置就是保存在subsys[]数组对应的实例上的，这样不同子系统就通过这种方式得到该进程应该进行哪些资源的管理和限制等工作。

整体CGroup框架的数据结构关系就是这些，总体其实主要包括两部分内容，首先完成资源组的关系建立，以及创建各子系统对应的核心数据结构，然后关联进程到不同资源组。

A.3.2　CGroup子系统初始化过程

下面我们结合部分源代码来详细分析CGroup的工作过程。CGroup框架的实现代码主要是kernel/cgroup.c这个文件，一些数据结构定义在include/linux/cgroup.h中。首先，我们从CGroup框架初始化开始介绍。

在内核启动函数start_kernel中（位于init/main.c文件中），我们会分别调用cgroup_init_early和cgroup_init函数进行CGroup的初始化工作，分别对应初始化的两个阶段：第一个阶段为调用cgroup_init_early，这是在内核各子系统初始化之前进行的，主要初始化好CGroup框架结构以及某些CGroup的子系统，例如CPU相关的CGroup子系统；第二个阶段调用cgroup_init初始化其他不需要提前构建的各个子系统，比如内存管理CGroup子系统及通用块管理的CGroup 子系统。整体初始化后形成的数据结构关系如图A-10所示。



图A-10　CGroup初始化

下面我们给出了CGroup初始化的相关代码：

```
// cgroup.c

1. int __init cgroup_init_early(void)
2. {
3.     struct cgroup_subsys *ss;
4.     int i;
5.
6.     atomic_set(&init_css_set.refcount, 1);
7.     INIT_LIST_HEAD(&init_css_set.cgrp_links);
8.     INIT_LIST_HEAD(&init_css_set.tasks);
9.     INIT_HLIST_NODE(&init_css_set.hlist);
10.    css_set_count = 1;
11.    init_cgroup_root(&cgroup_dummy_root);
12.    cgroup_root_count = 1;
13.    RCU_INIT_POINTER(init_task.cgroups, &init_css_set);
14.
15.    init_cgrp_cset_link.cset = &init_css_set;
16.    init_cgrp_cset_link.cgrp = cgroup_dummy_top;
17.    list_add(&init_cgrp_cset_link.cset_link, &cgroup_dummy_top->cset_links);
18.    list_add(&init_cgrp_cset_link.cgrp_link, &init_css_set.cgrp_links);
19.
20.    for_each_builtin_subsys(ss, i) {
21.        BUG_ON(!ss->name);
22.        BUG_ON(strlen(ss->name) > MAX_CGROUP_TYPE_NAMELEN);
23.        BUG_ON(!ss->css_alloc);
24.        BUG_ON(!ss->css_free);
25.        if (ss->subsys_id != i) {
26.            printk(KERN_ERR "cgroup: Subsys %s id == %d\n",
```

```
27.             ss->name, ss->subsys_id);
28.           BUG();
29.       }
30.
31.       if (ss->early_init)
32.           cgroup_init_subsys(ss);
33.   }
34.   return 0;
35.}
```

在上述代码中，init_css_set、init_task、init_cgrp_cset_link.cset以及cgroup_dummy_root都是全局变量。

第6行~第18行代码用于初始化这些数据结构的关系，第20行~第33行代码用于初始化CGroup里需要提前初始化的各个子系统。

在上述代码中，我们通过for_each_builtin_subsys来遍历系统内建的所有子系统，即系统定义好的各子系统的cgroup_subsys实例。for_each_builtin_subsys的定义如下：

```
// cgroup.c

#define for_each_builtin_subsys(ss, i)
    for ((i) = 0; (i) < CGROUP_BUILTIN_SUBSYS_COUNT &&
        (((ss) = cgroup_subsys[i]) || true); (i)++)
```

其中CGROUP_BUILTIN_SUBSYS_COUNT宏定义代表当前内核CGroup内建的子系统个数，cgroup_subsys是一个全局的指针数组，也定义在cgroup.c中，相关代码如下：

```
// cgroup.c

#define SUBSYS(_x) [_x ## _subsys_id] = &_x ## _subsys,
#define IS_SUBSYS_ENABLED(option) IS_BUILTIN(option)
static struct cgroup_subsys *cgroup_subsys[CGROUP_SUBSYS_COUNT] = {
    #include <linux/cgroup_subsys.h>
};
```

下面看下cgroup_subsys.h文件的内容：

```
// cgroup_subsys.h

#if IS_SUBSYS_ENABLED(CONFIG_CPUSETS)
SUBSYS(cpuset)
#endif

#if IS_SUBSYS_ENABLED(CONFIG_CGROUP_DEBUG)
SUBSYS(debug)
#endif
```

```
#if IS_SUBSYS_ENABLED(CONFIG_CGROUP_SCHED)
SUBSYS(cpu_cgroup)
#endif
...
```

结合在一起可以看出，这个指针数组的初始化内容就是各个子系统cgroup_subsys实例的地址。将宏展开，会得到如下的表达式：

```
static struct cgroup_subsys *cgroup_subsys[CGROUP_SUBSYS_COUNT] = {
    [cpuset_subsys_id] = &cpuset_subsys,
    [debug_subsys_id] = &debug_subsys,
    [cpu_cgroup_subsys_id] = &cpu_cgroup_subsys,
    ...
};
```

其中cpuset_subsys_id、debug_subsys_id等实际上是个全局的枚举值，这里相当于数组的索引号，其定义如下：

```
// cgroup.h

#define SUBSYS(_x) _x ## _subsys_id,
enum cgroup_subsys_id {
#define IS_SUBSYS_ENABLED(option) IS_BUILTIN(option)
#include <linux/cgroup_subsys.h>
#undef IS_SUBSYS_ENABLED
    CGROUP_BUILTIN_SUBSYS_COUNT,

    __CGROUP_SUBSYS_TEMP_PLACEHOLDER = CGROUP_BUILTIN_SUBSYS_COUNT - 1,

#define IS_SUBSYS_ENABLED(option) IS_MODULE(option)
#include <linux/cgroup_subsys.h>
#undef IS_SUBSYS_ENABLED
    CGROUP_SUBSYS_COUNT,
};
```

简单地说，就是把内建的及以模块形式加载的各个子系统都包含进来，相当于：

```
enum cgroup_subsys_id {
    cpuset_subsys_id,
    debug_subsys_id,
    cpu_cgroup_subsys_id
};
```

最后，我们具体看一下某个子系统的cgroup_subsys的定义。这里以CPU子系统为例，定义如下：

```
struct cgroup_subsys cpu_cgroup_subsys = {
    .name                = "cpu",
    .css_alloc           = cpu_cgroup_css_alloc,
    .css_free            = cpu_cgroup_css_free,
    .css_online          = cpu_cgroup_css_online,
    .css_offline         = cpu_cgroup_css_offline,
    .can_attach          = cpu_cgroup_can_attach,
    .attach              = cpu_cgroup_attach,
    .exit                = cpu_cgroup_exit,
    .subsys_id           = cpu_cgroup_subsys_id,
    .base_cftypes        = cpu_files,
    .early_init          = 1,
};
```

通过上面的代码，我们可以理解for_each_builtin_subsys其实就是简单地遍历所有内建的各个CGroup子系统的cgroup_subsys实例，而后根据是否设置了early_init的标志来判断是否需要进行提前的初始化工作。具体的初始化工作由cgroup_init_subsys函数完成，该函数的代码如下：

```
// cgroup.c

1. static void __init cgroup_init_subsys(struct cgroup_subsys *ss)
2. {
3.     struct cgroup_subsys_state *css;
4.
5.     printk(KERN_INFO "Initializing cgroup subsys %s\n", ss->name);
6.
7.     mutex_lock(&cgroup_mutex);
8.
9.     cgroup_init_cftsets(ss);
10.
11.    list_add(&ss->sibling, &cgroup_dummy_root.subsys_list);
12.    ss->root = &cgroup_dummy_root;
13.    css = ss->css_alloc(cgroup_css(cgroup_dummy_top, ss));
14.
15.    BUG_ON(IS_ERR(css));
16.    init_css(css, ss, cgroup_dummy_top);
17.
18.    init_css_set.subsys[ss->subsys_id] = css;
19.
20.    need_forkexit_callback |= ss->fork || ss->exit;
21.
22.    BUG_ON(!list_empty(&init_task.tasks));
23.
24.    BUG_ON(online_css(css));
25.
```

```
26.    mutex_unlock(&cgroup_mutex);
27.
28.    BUG_ON(ss->module);
29.}
```

第9行代码进行子系统的文件初始化，对应的就是我们挂载一个子系统时系统自动为我们生成的那些文件。这些文件实际上定义在cgroup_subsys数据结构中。

第11行～第12行代码用于建立cgroup_dummy_root与各子系统的cgroup_subsys实例的数据结构的关联。

第13行代码比较关键。这里通过各子系统定义的cgroup_subsys实例来创建各自的核心数据结构，即cgroup_subsys_state的实例。

对应的CPU子系统就是调用cpu_cgroup_css_alloc函数并返回task_group实例，其中第一个成员是cgroup_subsys_state，然后通过该成员与CGroup框架其他核心数据结构建立关系。

第16行代码到函数结束基本上都是一些数据结构的初始化和关联操作，可以参考前面的数据结构关系图，这里不再赘述。

看完cgroup_init_early函数后，我们再来看下cgroup_init函数。这个函数比较长，这里就不列出来了。总体上该函数的核心功能有两个：第一个和前面的cgroup_init_early基本一样，会初始化其他没有提前初始化的各子系统，即遍历没有定义early_init的各cgroup_subsys实例，并进行cgroup_init_subsys操作。

第二个重要工作是向内核注册CGroup文件系统，这通过下面的代码进行：

```
err = register_filesystem(&cgroup_fs_type);
```

内核有一个file_system_type类型的全局变量file_systems，它以链表的形式保存了所有向内核注册过的文件系统。file_system_type主要提供了后续装载该文件系统的重要回调函数.mount，通过该函数来完成文件系统的装载工作。

我们看一下cgroup_fs_type的定义：

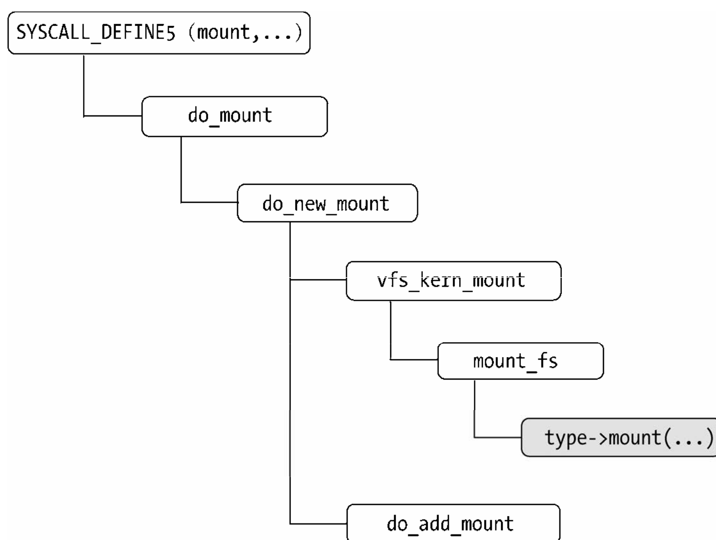
```
static struct file_system_type cgroup_fs_type = {
    .name = "cgroup",
    .mount = cgroup_mount,
    .kill_sb = cgroup_kill_sb,
};
```

对我们来说，最重要的是cgroup_mount函数，该函数就是后续我们装载CGroup文件系统时会调用的核心实现方法。至此，CGroup框架的初始化工作完成。下面简单总结下CGroup框架初始化所做的工作。

- 创建并初始化好cgroup_dummy_root、nit_css_set、cgroup_dummy_top以及各子系统cgroup_subsys数据结构及关联关系。
- 通过各子系统定义的cgroup_subsys实例，调用其css_alloc成员函数完成各子系统核心数据结构的建立，并通过其首个成员cgroup_subsys_state挂载到init_css_set上，并将init_css_set与init_task建立关联。
- 通过register_filesystem向内核注册CGroup文件系统。

A.3.3　CGroup文件系统的挂载实现

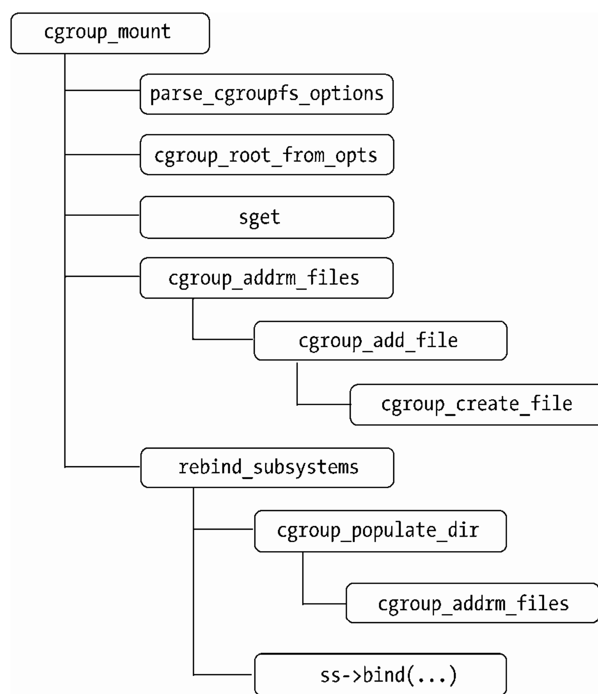
通过前面文件系统的注册，我们知道当CGroup文件系统被装载时，内核最终会调用cgroup_mount函数进行实际的工作。我们在看这个函数的实现前，先来了解一下VFS挂载文件系统所经过的函数调用栈。总体过程仍然是机制和策略分离的，VFS提供通用的一个挂载文件系统的方法，各个文件系统各自的初始化和设置工作都是通过file_system_type的mount回调函数完成的，VFS挂载文件系统通用部分所做的事情主要是初始化文件系统的super_block、文件系统根目录inode以及对应dentry的关系。主要的函数调用栈如图A-11所示。



图A-11　CGroup文件系统挂载过程中的函数调用

进入mount系统调用后，内核首先会根据传入的参数确定要挂载的文件系统类型，即找到对应的file_system_type数据结构实例，然后根据挂载的flags参数确定执行哪种挂载形式（例如是重新挂载还是进行全新的挂载，这里我们假设是全新的挂载），最终内核会调用file_system_type实例type的mount回调方法，这就是我们之前注册CGroup文件系统时提供的cgroup_mount函数。

cgroup_mount函数调用栈如图A-12所示。



图A-12　cgroup_mount函数调用栈

首先，cgroup_mount调用parse_cgroupfs_options函数解析用户mount命令对应的flags参数，即mount命令中-o选项对应的内容，解析结果保存在cgroup_sb_opts结构体中。该结构体的定义如下：

```
struct cgroup_sb_opts {
    unsigned long subsys_mask;
    unsigned long flags;
    char *release_agent;
    bool cpuset_clone_children;
    char *name;
    bool none;

    struct cgroupfs_root *new_root;

};
```

下面简要介绍一下该结构体中各个成员的含义。

- subsys_mask：它是一个位图，每个位代表一个子系统是否需要装载。根据用户mount -o指定的子系统，来将对应位设置成1。

- Flags：它代表除了挂载的子系统外的其他标记。例如，可以指定noprefix，代表不对CGroup文件系统下自动生成的文件加子系统的前缀。例如，默认的blkio子系统会生成blkio.weight文件，如果装载文件系统时指定了noprefix参数，则对应的文件名就是weight。
- release_agent：其中存储的命令会在某个CGroup目录中最后一个进程被移除时自动执行。这些命令可以在装载CGroup文件系统时在参数中指定，也可以在装载CGroup文件系统后向release_agent文件写入。
- cpuset_clone_children：如果该成员为true，则代表一个新的cpuset初始化时，会自动继承父cpuset的配置。
- name：该参数指我们可以为装载实例命名，一般很少会用到。
- none：标记我们是否装载了一个空的CGroup文件系统。
- new_root：用于保存后续生成的cgroupfs_root实例。

CGroup的装载选项被保存到了cgroup_sb_opts结构体的实例后，系统会根据这个结果生成对应的cgroupfs_root实例，这可以通过cgroup_root_from_opts函数完成，其主要流程是将cgroup_sb_opts中保存的值复制到cgroupfs_root实例对应的成员上以及其他一些基本的初始化工作。

之后的一个重要工作是调用VFS提供的通用函数sget来初始化CGroup文件系统的super_block。理解这个函数对于我们理解CGroup的装载过程还是很重要的，下面我们先来看看这个函数，其代码如下：

```
1.struct super_block *sget(struct file_system_type *type,
2.                    int (*test)(struct super_block *,void *),
3.                    int (*set)(struct super_block *,void *),
4.                    int flags,
5.                    void *data)
6.{
7.     struct super_block *s = NULL;
8.     struct super_block *old;
9.     int err;
10.
11.retry:
12.    spin_lock(&sb_lock);
13.    if(test) {
14.        hlist_for_each_entry(old, &type->fs_supers, s_instances) {
15.            if (!test(old, data))
16.                continue;
17.            if (!grab_super(old))
18.                goto retry;
19.            if (s) {
20.                up_write(&s->s_umount);
21.                destroy_super(s);
22.                s = NULL;
```

```
23.             }
24.             return old;
25.         }
26.     }
27.     if (!s) {
28.         spin_unlock(&sb_lock);
29.         s = alloc_super(type, flags);
30.         if (!s)
31.             return ERR_PTR(-ENOMEM);
32.         goto retry;
33.     }
34.
35.     err = set(s, data);
36.     if (err) {
37.         spin_unlock(&sb_lock);
38.         up_write(&s->s_umount);
39.         destroy_super(s);
40.         return ERR_PTR(err);
41.     }
42.     s->s_type = type;
43.     strlcpy(s->s_id, type->name, sizeof(s->s_id));
44.     list_add_tail(&s->s_list, &super_blocks);
45.     hlist_add_head(&s->s_instances, &type->fs_supers);
46.     spin_unlock(&sb_lock);
47.     get_filesystem(type);
48.     register_shrinker(&s->s_shrink);
49.     return s;
50.}
51.
52.EXPORT_SYMBOL(sget);
```

第13行～第26行代码遍历传入文件系统file_system_type实例的fs_supers链表，该链表保存了具体一个文件系统实例下面创建的所有super_block实例，这里就是得到我们注册的CGroup文件系统实例下所有的super_block，并根据传入的test回调函数来判断是否可以复用之前的super_block。如果该回调函数返回true，并且后面第17行代码通过grab_super增加该super_block实例的引用成功，则在第24行代码中直接返回已经存在的super_block实例。

我们传入的test回调函数实际上是cgroup_test_super，它是在调用sget时传入的：

```
/* 定位到一个已经存在的super_block或新建一个super_block */
sb = sget(fs_type, cgroup_test_super, cgroup_set_super, 0, &opts);
```

接下来我们看一下cgroup_test_super这个函数的实现：

```
1. static int cgroup_test_super(struct super_block *sb, void *data)
2. {
3.     struct cgroup_sb_opts *opts = data;
4.     struct cgroupfs_root *root = sb->s_fs_info;
```

```
5.
6.     if (opts->name && strcmp(opts->name, root->name))
7.         return 0;
8.
9.     if ((opts->subsys_mask || opts->none)
10.         && (opts->subsys_mask != root->subsys_mask))
11.         return 0;
12.
13.     return 1;
14.}
```

从前面的sget函数得知，如果该函数返回1，代表可以复用已经存在的super_block实例。该函数主要是两条判断语句。

第6行代码判断挂载时是否指定了名字，即命令中是否包含-o name=xxx选项。如果指定了名字，则会判断当前super_block实例对应的名字是否完全相同。如果名字不匹配，则直接返回0，即不能复用该super_block。

第9行～第11行代码通过subsys_mask位图来判断当前mount命令指定的子系统是否与传入的super_block实例的子系统完全相同，如果不同，则直接返回0。

整个函数的功能其实就是检测是否可以复用函数传入的super_block实例。如果mount选项指定了名字，则名字必须完全匹配，并且mount指定的所有子系统也必须完全一样，这样就可以复用原来已经存在的super_block实例了。

我们继续回到sget函数中，如果没有可以复用的super_block，则代码向下继续执行。

第27行～第33行代码通过alloc_super函数分配super_block。

第35行代码通过函数传入的set回调函数来初始化我们刚刚分配的super_block数据结构。传入的回调函数实际上是cgroup_set_super，下面我们来看一下这个函数，其代码如下：

```
1. static int cgroup_set_super(struct super_block *sb, void *data)
2. {
3.     int ret;
4.     struct cgroup_sb_opts *opts = data;
5.
6.     if (!opts->new_root)
7.         return -EINVAL;
8.
9.     BUG_ON(!opts->subsys_mask && !opts->none);
10.
11.     ret = set_anon_super(sb, NULL);
12.     if (ret)
13.         return ret;
14.
```

```
15.     sb->s_fs_info = opts->new_root;
16.     opts->new_root->sb = sb;
17.
18.     sb->s_blocksize = PAGE_CACHE_SIZE;
19.     sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
20.     sb->s_magic = CGROUP_SUPER_MAGIC;
21.     sb->s_op = &cgroup_ops;
22.
23.     return 0;
24.}
```

第6行～第9行代码做一些基本的验证和判断，主要是确认之前需要初始化的几个成员。

第11行代码通过set_anon_super函数来分配super_block实例对应的虚拟设备号，并将其保存在super_block的s_dev成员中，这里不再详细展开。

第15行代码将我们新分配的cgroupfs_root数据结构放入super_block的s_fs_info成员中。该成员用于保存文件系统的私有核心数据结构。

第15行～第20行代码进行super_block其他一些成员的默认初始化工作。第21行代码为super_block注册了super_operations的回调函数为cgroup_ops。

继续回到前面的sget函数，第42行代码到结束主要将新分配和初始化好的super_block实例加入到全局的super_blocks链表中，同时将其加入到CGroup文件系统file_system_type的fs_supers链表中。后面的register_shrinker函数主要用于注册一些回调方法，在内存紧张时可以通过该接口帮助内存子系统回收一些缓存资源。

我们回到cgroup_mount函数的主流程中，该函数后续通过调用cgroup_addrm_files来生成CGroup框架的相关文件，例如tasks文件等。cgroup_addrm_files函数的定义如下：

```
1. static int cgroup_addrm_files(struct cgroup *cgrp, struct cftype cfts[],
2.                               bool is_add)
3. {
4.     struct cftype *cft;
5.     int ret;
6.
7.     lockdep_assert_held(&cgrp->dentry->d_inode->i_mutex);
8.     lockdep_assert_held(&cgroup_mutex);
9.
10.    for (cft = cfts; cft->name[0] != '\0'; cft++) {
11.        if ((cft->flags & CFTYPE_INSANE) && cgroup_sane_behavior(cgrp))
12.            continue;
13.        if ((cft->flags & CFTYPE_NOT_ON_ROOT) && !cgrp->parent)
14.            continue;
15.        if ((cft->flags & CFTYPE_ONLY_ON_ROOT) && cgrp->parent)
16.            continue;
```

```
17.
18.         if (is_add) {
19.             ret = cgroup_add_file(cgrp, cft);
20.             if (ret) {
21.                 pr_warn("cgroup_addrm_files: failed to add %s, err=%d\n",
22.                     cft->name, ret);
23.                 return ret;
24.             }
25.         } else {
26.             cgroup_rm_file(cgrp, cft);
27.         }
28.     }
29.     return 0;
30.}
```

该函数的功能是遍历我们传入的cfts数组，它根据is_add标志依次调用cgroup_add_file或者cgroup_rm_file函数来创建或删除cgroup文件。这里我们传入的is_add参数为true，也就是说需要创建cfts数组里的所有文件。cfts数组在调用该函数时指定的是cgroup_base_files，下面我们看一下这个数组的内容：

```
static struct cftype cgroup_base_files[] = {
    {
        .name = "cgroup.procs",
        .open = cgroup_procs_open,
        .write_u64 = cgroup_procs_write,
        .release = cgroup_pidlist_release,
        .mode = S_IRUGO | S_IWUSR,
    },
    {
        .name = "cgroup.event_control",
        .write_string = cgroup_write_event_control,
        .mode = S_IWUGO,
    },
    {
        .name = "cgroup.clone_children",
        .flags = CFTYPE_INSANE,
        .read_u64 = cgroup_clone_children_read,
        .write_u64 = cgroup_clone_children_write,
    },
    {
        .name = "cgroup.sane_behavior",
        .flags = CFTYPE_ONLY_ON_ROOT,
        .read_seq_string = cgroup_sane_behavior_show,
    },
    {
        .name = "tasks",
        .flags = CFTYPE_INSANE,
        .open = cgroup_tasks_open,
```

```
        .write_u64 = cgroup_tasks_write,
        .release = cgroup_pidlist_release,
        .mode = S_IRUGO | S_IWUSR,
    },
    {
        .name = "notify_on_release",
        .flags = CFTYPE_INSANE,
        .read_u64 = cgroup_read_notify_on_release,
        .write_u64 = cgroup_write_notify_on_release,
    },
    {
        .name = "release_agent",
        .flags = CFTYPE_INSANE | CFTYPE_ONLY_ON_ROOT,
        .read_seq_string = cgroup_release_agent_show,
        .write_string = cgroup_release_agent_write,
        .max_write_len = PATH_MAX,
    },
    {}
};
```

从上面的定义能够看出，该数组定义了CGroup框架本身所包含的一些配置文件以及相关的读写回调函数。下面我们看一下cftype数据结构的定义：

```
struct cftype {
    char name[MAX_CFTYPE_NAME];
    int private;
    umode_t mode;
    size_t max_write_len;
    unsigned int flags;
    struct cgroup_subsys *ss;

    int (*open)(struct inode *inode, struct file *file);
    ssize_t (*read)(struct cgroup_subsys_state *css, struct cftype *cft,
                struct file *file,
                char __user *buf, size_t nbytes, loff_t *ppos);
    u64 (*read_u64)(struct cgroup_subsys_state *css, struct cftype *cft);
    s64 (*read_s64)(struct cgroup_subsys_state *css, struct cftype *cft);
    int (*read_map)(struct cgroup_subsys_state *css, struct cftype *cft,
                struct cgroup_map_cb *cb);
    int (*read_seq_string)(struct cgroup_subsys_state *css,
                       struct cftype *cft, struct seq_file *m);

    ssize_t (*write)(struct cgroup_subsys_state *css, struct cftype *cft,
                 struct file *file,
                 const char __user *buf, size_t nbytes, loff_t *ppos);
    int (*write_u64)(struct cgroup_subsys_state *css, struct cftype *cft,u64 val);
    int (*write_s64)(struct cgroup_subsys_state *css, struct cftype *cft,s64 val);
    int (*write_string)(struct cgroup_subsys_state *css, struct cftype *cft,
                    const char *buffer);
```

```
    int (*trigger)(struct cgroup_subsys_state *css, unsigned int event);

    int (*release)(struct inode *inode, struct file *file);

    int (*register_event)(struct cgroup_subsys_state *css,
                          struct cftype *cft, struct eventfd_ctx *eventfd,
                          const char *args);
    void (*unregister_event)(struct cgroup_subsys_state *css,
                             struct cftype *cft,
                             struct eventfd_ctx *eventfd);
};
```

该数据结构包含的成员比较多，我们就不一一解释了。实际上，CGroup文件系统中每个自动生成的文件都是该数据结构类型的一个实例。该数据结构定义了这些文件的文件名、读写回调函数以及注册相关事件的回调函数等。

我们继续看看cgroup_add_file函数的实现：

```
1. static int cgroup_add_file(struct cgroup *cgrp, struct cftype *cft)
2. {
3.     struct dentry *dir = cgrp->dentry;
4.     struct cgroup *parent = __d_cgrp(dir);
5.     struct dentry *dentry;
6.     struct cfent *cfe;
7.     int error;
8.     umode_t mode;
9.     char name[MAX_CGROUP_TYPE_NAMELEN + MAX_CFTYPE_NAME + 2] = { 0 };
10.
11.    if (cft->ss && !(cft->flags & CFTYPE_NO_PREFIX) &&
12.       !(cgrp->root->flags & CGRP_ROOT_NOPREFIX)) {
13.       strcpy(name, cft->ss->name);
14.       strcat(name, ".");
15.    }
16.    strcat(name, cft->name);
17.
18.    BUG_ON(!mutex_is_locked(&dir->d_inode->i_mutex));
19.
20.    cfe = kzalloc(sizeof(*cfe), GFP_KERNEL);
21.    if (!cfe)
22.        return -ENOMEM;
23.
24.    dentry = lookup_one_len(name, dir, strlen(name));
25.    if (IS_ERR(dentry)) {
26.        error = PTR_ERR(dentry);
27.        goto out;
28.    }
29.
30.    cfe->type = (void *)cft;
```

```
31.    cfe->dentry = dentry;
32.    dentry->d_fsdata = cfe;
33.    simple_xattrs_init(&cfe->xattrs);
34.
35.    mode = cgroup_file_mode(cft);
36.    error = cgroup_create_file(dentry, mode | S_IFREG, cgrp->root->sb);
37.    if (!error) {
38.        list_add_tail(&cfe->node, &parent->files);
39.        cfe = NULL;
40.    }
41.    dput(dentry);
42.out:
43.    kfree(cfe);
44.    return error;
45.}
```

第11行～第16行代码根据该文件所在的子系统以及文件本身的名字，拼接出实际要生成文件的名字。如果挂载时指定了noprefix或者文件定义本身标记了noprefix，则文件名中不包含子系统的名字。

第20行代码分配一个cfent类型的变量cfe，该类型代表的含义是CGroup文件入口，实际相当于是通用目录项dentry数据结构的扩展，cfe存储在文件对应的目录项dentry的d_fsdata域，该域用来存放具体文件系统目录项dentry的私有数据结构。第30行~第33行代码就是实现这个过程的。

第24行代码通过lookup_one_len查找并创建待生成文件的dentry数据结构。在VFS中，dentry抽象了整个文件系统目录的层次结构。

第36行代码通过cgroup_create_file来创建和初始化文件对应的索引结点。创建成功后，第38行代码将cfe加入到父cgroup的files链表中。

我们再看一下cgroup_create_file函数，其代码如下：

```
1. static int cgroup_create_file(struct dentry *dentry, umode_t mode,
2.                               struct super_block *sb)
3. {
4.    struct inode *inode;
5.
6.    if (!dentry)
7.        return -ENOENT;
8.    if (dentry->d_inode)
9.        return -EEXIST;
10.
11.    inode = cgroup_new_inode(mode, sb);
12.    if (!inode)
13.        return -ENOMEM;
14.
15.    if (S_ISDIR(mode)) {
```

```
16.         inode->i_op = &cgroup_dir_inode_operations;
17.         inode->i_fop = &simple_dir_operations;
18.
19.         inc_nlink(inode);
20.         inc_nlink(dentry->d_parent->d_inode);
21.
22.         WARN_ON_ONCE(!mutex_trylock(&inode->i_mutex));
23.     } else if (S_ISREG(mode)) {
24.         inode->i_size = 0;
25.         inode->i_fop = &cgroup_file_operations;
26.         inode->i_op = &cgroup_file_inode_operations;
27.     }
28.     d_instantiate(dentry, inode);
29.     dget(dentry);
30.     return 0;
31.}
```

第11行代码创建inode数据结构。

第15行～第27行代码根据当前文件的类型是目录还是普通文件来初始化inode数据结构。这里主要是指定i_fop和i_op两个成员，它们分别对应file_operations和inode_operations类型。对于目录，我们主要关心i_op成员，里面定义了操作inode节点的特定回调方法，例如创建CGroup目录等。对于文件，我们主要关心i_fop成员，里面定义了文件读写的特定回调方法。

第28行代码将分配的inode数据结构与文件的dentry建立关联。

我们再详细看一下cgroup_dir_inode_operations及cgroup_file_operations的定义：

```
static const struct file_operations cgroup_file_operations = {
    .read = cgroup_file_read,
    .write = cgroup_file_write,
    .llseek = generic_file_llseek,
    .open = cgroup_file_open,
    .release = cgroup_file_release,
};

static const struct inode_operations cgroup_dir_inode_operations = {
    .lookup = simple_lookup,
    .mkdir = cgroup_mkdir,
    .rmdir = cgroup_rmdir,
    .rename = cgroup_rename,
    .setxattr = cgroup_setxattr,
    .getxattr = cgroup_getxattr,
    .listxattr = cgroup_listxattr,
    .removexattr = cgroup_removexattr,
};
```

当读写CGroup文件系统中自动生成的那些文件时，内核VFS最终会调用cgroup_file_operations

里定义的那些读写函数。当创建或者删除目录时，内核最终会调用cgroup_dir_inode_operations里的那些回调函数。后面我们再详细讨论这个过程。

现在继续回到主流程。cgroup_mount最后进行的步骤是rebind_subsystems，该函数的主要功能是生成各子系统的相关配置文件，初始化好cgroupfs_root的top_cgroup成员与mount上的子系统cgroup_subsys及cgroup_subsys_state的关系（这部分工作主要通过调用cgroup-populate-dir函数完成），最终调用各个子系统的cgroup_subsys的bind方法来完成子系统特定的逻辑。

下面我们看一下cgroup_populate_dir的代码：

```
1. static int cgroup_populate_dir(struct cgroup *cgrp, unsigned long subsys_mask)
2. {
3.     struct cgroup_subsys *ss;
4.     int i, ret = 0;
5.
6.     for_each_subsys(ss, i) {
7.         struct cftype_set *set;
8.
9.         if (!test_bit(i, &subsys_mask))
10.            continue;
11.
12.        list_for_each_entry(set, &ss->cftsets, node) {
13.            ret = cgroup_addrm_files(cgrp, set->cfts, true);
14.            if (ret < 0)
15.                goto err;
16.        }
17.    }
18.    return 0;
19.err:
20.    cgroup_clear_dir(cgrp, subsys_mask);
21.    return ret;
22.}
```

这段代码主要是一个两层循环：外层是第7行～第17行代码，用于遍历所有的子系统；内层是第12行～第16行代码，用于遍历cgroup_subsys数据结构中cftsets链表的所有成员。该链表的节点数据类型如下：

```
struct cftype_set {
    struct list_head   node;
    struct cftype      *cfts;
};
```

其中node成员用于链接cgroup_subsys的cftsets链表。cfts成员实际上是一组cftype实例，对应一组cgroup的配置文件，该数据类型前面已经提到，这里不多解释了。每个子系统的cgroup配置文件都是挂载在cgroup_subsys的cftsets链表上的。每个cftype_set实例对应一组相关的文件，不同组之间的文件一般是不同的策略。内核通过cgroup_add_cftypes函数向cgroup_subsys加入配置文件。

cgroup_populate_dir函数实际上就是遍历所有子系统的配置文件，然后通过cgroup_addrm_files来增删对应的文件，我们这里传入的参数true代表增加文件。cgroup文件系统中的大部分文件都是在这里创建出来的。

最后，我们看一下rebind_subsystems函数剩余部分的主要代码：

```
1. static int rebind_subsystems(struct cgroupfs_root *root,
2.                              unsigned long added_mask, unsigned removed_mask)
3. {
4.     ...
5.     for_each_subsys(ss, i) {
6.         unsigned long bit = 1UL << i;
7.
8.         if (bit & added_mask) {
9.             BUG_ON(cgroup_css(cgrp, ss));
10.            BUG_ON(!cgroup_css(cgroup_dummy_top, ss));
11.            BUG_ON(cgroup_css(cgroup_dummy_top, ss)->cgroup != cgroup_dummy_top);
12.
13.            rcu_assign_pointer(cgrp->subsys[i],
14.                               cgroup_css(cgroup_dummy_top, ss));
15.            cgroup_css(cgrp, ss)->cgroup = cgrp;
16.
17.            list_move(&ss->sibling, &root->subsys_list);
18.            ss->root = root;
19.            if (ss->bind)
20.               ss->bind(cgroup_css(cgrp, ss));
21.
22.            root->subsys_mask |= bit;
23.        } else if (bit & removed_mask) {
24.            ...
25.        }
26.    }
27.
28.    ...
29.
30.}
```

最后一部分主要是第5行～第26行代码，用于遍历所有子系统，根据前面初始化好的cgroupfs_root的subsys_mask成员执行相关子系统的特定逻辑，这里函数传入的参数removed_mask为0，所以我们直接忽略第23行代码后面的分支语句。

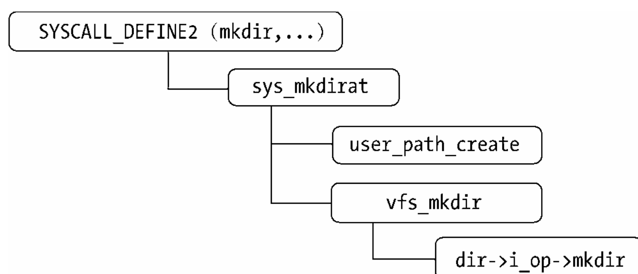
第13行～第15行代码主要将各个子系统对应的实例与挂载上来的CGroup文件系统的相关实例进行关联。

第17行～第18行代码将相应的cgroup_subsys实例挂载到cgroupfs_root的subsys_list链表头中，并建立反向指向cgroupfs_root的指针。

第19行～第20行代码，根据cgroup_subsys是否指定bind回调函数来执行子系统特定的代码。我们后面再讨论子系统特定的实现逻辑。

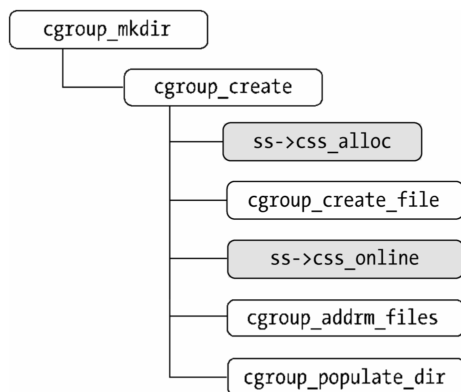
整个cgroup_mount函数的执行过程已分析完毕，下面我们看一下在CGroup文件系统下创建一个目录内核时对应的代码执行路径。在CGroup文件系统下创建目录内核时，首先执行的是VFS部分的通用逻辑代码，其代码调用关系如图A-13所示。

内核先通过user_path_create做路径查找并创建好待创建目录的inode数据结构，之后在vfs_mkdir中调用该inode的mkdir回调方法，这里对应的就是cgroup_mkdir。



图A-13　VFS创建目录调用关系图

cgroup_mkdir函数的执行路径如图A-14所示，其代码如下所示。



图A-14　cgroup_mkdir函数的执行路径

我们看一下cgroup_mkdir的代码：

```
1. static int cgroup_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode)
2. {
3.     struct cgroup *c_parent = dentry->d_parent->d_fsdata;
4.
5.     return cgroup_create(c_parent, dentry, mode | S_IFDIR);
6. }
```

第3行代码通过dentry的d_parent成员找到待创建目录的父目录结构，之后通过父目录dentry的d_fsdata成员得到cgroup的实例。

第5行代码返回cgroup_create函数调用的结果。由于cgroup_create函数的代码非常多，所以这里只罗列部分重要的函数调用逻辑。

首先创建相应的cgroup数据结构实例，然后进行必要的一些成员初始化工作，相关代码如下：

```
1. static long cgroup_create(struct cgroup *parent, struct dentry *dentry,
2.                           umode_t mode)
3. {
4.     ...
5.     atomic_inc(&sb->s_active);
6.
7.     init_cgroup_housekeeping(cgrp);
8.
9.     dentry->d_fsdata = cgrp;
10.    cgrp->dentry = dentry;
11.
12.    cgrp->parent = parent;
13.    cgrp->dummy_css.parent = &parent->dummy_css;
14.    cgrp->root = parent->root;
15.
16.    if (notify_on_release(parent))
17.        set_bit(CGRP_NOTIFY_ON_RELEASE, &cgrp->flags);
18.
19.    if (test_bit(CGRP_CPUSET_CLONE_CHILDREN, &parent->flags))
20.        set_bit(CGRP_CPUSET_CLONE_CHILDREN, &cgrp->flags);
21.        ...
22.}
```

第5行代码增加cgroup文件系统super_block的引用计数，防止cgroup文件系统被卸载。

第7行代码调用init_cgroup_housekeeping初始化cgroup的一些成员，这里不详细列出。

第9行～第10行代码将VFS的dentry与相应的cgroup数据结构之间建立关联。

第12行～第14行代码建立cgroup的层级关系。

第16行～第20行代码根据保存在父cgroup实例上的文件系统装载标志来设置相应新建的cgroup实例的flags成员。

随后，该函数进行如下逻辑：

```
...
for_each_root_subsys(root, ss) {
    struct cgroup_subsys_state *css;
```

```
        css = ss->css_alloc(cgroup_css(parent, ss));
        if (IS_ERR(css)) {
            err = PTR_ERR(css);
            goto err_free_all;
        }
        css_ar[ss->subsys_id] = css;

        err = percpu_ref_init(&css->refcnt, css_release);
        if (err)
            goto err_free_all;

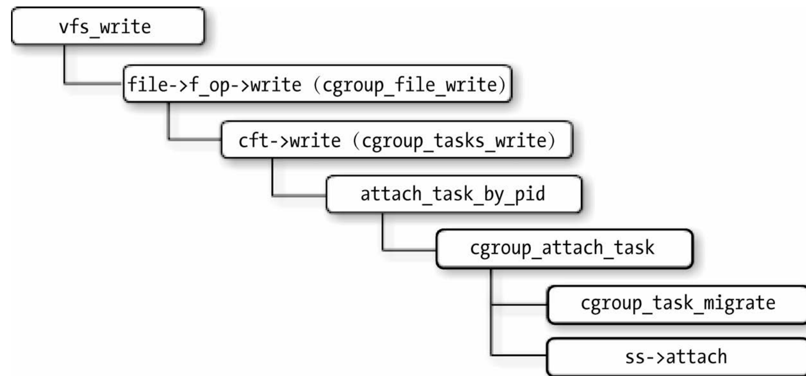
        init_css(css, ss, cgrp);
    }
    ...
```

这段代码遍历该装载实例对应的所有子系统，然后分别调用子系统cgroup_subsys的css_alloc函数进行子系统核心数据结构的分配和创建工作，并通过init_css函数来进行一些成员的初始化工作。init_css函数的代码如下：

```
1. static void init_css(struct cgroup_subsys_state *css, struct cgroup_subsys *ss,
2.                      struct cgroup *cgrp)
3. {
4.     css->cgroup = cgrp;
5.     css->ss = ss;
6.     css->flags = 0;
7.
8.     if (cgrp->parent)
9.         css->parent = cgroup_css(cgrp->parent, ss);
10.    else
11.        css->flags |= CSS_ROOT;
12.
13.    BUG_ON(cgroup_css(cgrp, ss));
14.}
```

这段代码主要是初始化子系统核心数据结构cgroup_subsys_state实例与当前新建的cgroup以及cgroup_subsys的关系，并建立好cgroup_subsys_state的父子关系。

在CGroup文件系统下创建目录的逻辑基本就是这些，下面我们来看一下将某个进程加入到CGroup资源组内核所做的工作。前面已经介绍过将指定进程加入资源组的方法是将该进程的pid写入相应资源组目录的tasks文件，整个文件写入从VFS开始的代码调用栈如图A-15所示。



图A-15　VFS文件写入调用关系图

代码进入VFS后，首先根据打开的文件file数据结构来调用f_op->write函数。这个函数指针实际上是在文件被打开时，从inode->i_fop那里复制过来的，这里对应的回调函数是cgroup_file_write。下面我们看看cgroup_file_write的代码：

```
1. static ssize_t cgroup_file_write(struct file *file, const char __user *buf,
2.                                  size_t nbytes, loff_t *ppos)
3. {
4.     struct cfent *cfe = __d_cfe(file->f_dentry);
5.     struct cftype *cft = __d_cft(file->f_dentry);
6.     struct cgroup_subsys_state *css = cfe->css;
7.
8.     if (cft->write)
9.         return cft->write(css, cft, file, buf, nbytes, ppos);
10.    if (cft->write_u64 || cft->write_s64)
11.        return cgroup_write_X64(css, cft, file, buf, nbytes, ppos);
12.    if (cft->write_string)
13.        return cgroup_write_string(css, cft, file, buf, nbytes, ppos);
14.    if (cft->trigger) {
15.        int ret = cft->trigger(css, (unsigned int)cft->private);
16.        return ret ? ret : nbytes;
17.    }
18.    return -EINVAL;
19.}
```

现在我们先将第4行~第5行代码的两个宏定义列出来：

```
static inline struct cfent *__d_cfe(struct dentry *dentry)
{
    return dentry->d_fsdata;
}

static inline struct cftype *__d_cft(struct dentry *dentry)
{
```

```
    return __d_cfe(dentry)->type;
}
```

可以看到，第4行～第5行代码主要是通过当前文件对应的dentry数据结构取得之前存入在dentry的d_fsdata成员中的cftype实例。根据前面cgroup_base_files的定义，我们得到了当前tasks文件的cft->write回调函数，这里对应的是cgroup_tasks_write函数，该函数直接调用attach_task_by_pid函数，相关代码如下：

```
static int cgroup_tasks_write(struct cgroup_subsys_state *css,
                              struct cftype *cft, u64 pid)
{
    return attach_task_by_pid(css->cgroup, pid, false);
}
```

这里特殊说明一下，attach_task_by_pid函数的第三个参数false代表我们传入的pid实际上是一个进程的pid，而不是tgid。

attach_task_by_pid函数内部会根据传入的pid来查找到对应进程的task_struct数据结构,随后调用cgroup_attach_task函数。

cgroup_attach_task函数比较长，这里我们一部分一部分地讨论。首先，函数会根据传入的进程ID来收集涉及的所有线程并将其保存到一个flex_array类型的容器中，相关代码如下：

```
  ...
1.    do {
2.        struct task_and_cgroup ent;
3.
4.        if (tsk->flags & PF_EXITING)
5.            goto next;
6.
7.        BUG_ON(i >= group_size);
8.        ent.task = tsk;
9.        ent.cgrp = task_cgroup_from_root(tsk, root);
10.       if (ent.cgrp == cgrp)
11.           goto next;
12.       retval = flex_array_put(group, i, &ent, GFP_ATOMIC);
13.       BUG_ON(retval != 0);
14.       i++;
15.   next:
16.       if (!threadgroup)
17.           break;
18.   } while_each_thread(leader, tsk);
19....
```

在上述代码中，while_each_thread宏用来遍历传入的tsk进程包含的所有线程。上述代码使用task_and_cgroup数据结构来保存对应线程的数据结构task_struct以及线程原来所属的cgroup。第12行代码将task_and_cgroup实例放入flex_array容器中。

在这之后，cgroup_attach_task函数会遍历所有的cgroup_subsys实例并调用can_attach函数来判断是否可以将指定进程加入该资源组，相关代码如下：

```
1. for_each_root_subsys(root, ss) {
2.     struct cgroup_subsys_state *css = cgroup_css(cgrp, ss);
3.
4.     if (ss->can_attach) {
5.         retval = ss->can_attach(css, &tset);
6.         if (retval) {
7.             failed_ss = ss;
8.             goto out_cancel_attach;
9.         }
10.    }
11.}
```

如果某个子系统对应的can_attach函数返回非零值，则代表进程不可以加入该资源组。第5行代码调用了can_attach函数，该函数的第一个参数是该资源组的cgroup_subsys_state实例，第二个参数是前面收集到的线程ID列表。

在第6行~第9行代码中，如果发现某个子系统不允许指定的线程加入，则整体操作失败，退出cgroup_attach_task函数。

继续回到主流程，代码后续会初始化好待加入线程的css_set数据结构，相关代码如下：

```
1. for (i = 0; i < group_size; i++) {
2.     struct css_set *old_cset;
3.
4.     tc = flex_array_get(group, i);
5.     old_cset = task_css_set(tc->task);
6.     tc->cset = find_css_set(old_cset, cgrp);
7.     if (!tc->cset) {
8.         retval = -ENOMEM;
9.         goto out_put_css_set_refs;
10.    }
11.}
```

该段代码主要用于遍历前面收集到的所有线程，然后为线程分配对应的css_set数据结构。第6行代码中，find_css_set函数会判断当前进程的css_set是否可以复用系统已经存在的css_set数据结构。例如，如果两个进程所加入的CGroup资源组的所有子系统的cgroup_subsys_state实例是完全相同的，则可以共享css_set数据结构，否则会分配新的css_set实例。

继续回到cgroup_attach_task函数，剩下部分的主要代码逻辑如下：

```
1. for (i = 0; i < group_size; i++) {
2.     tc = flex_array_get(group, i);
3.     cgroup_task_migrate(tc->cgrp, tc->task, tc->cset);
4. }
```

```
5.
6. for_each_root_subsys(root, ss) {
7.     struct cgroup_subsys_state *css = cgroup_css(cgrp, ss);
8.
9.     if (ss->attach)
10.        ss->attach(css, &tset);
11.}
```

第1行~第3行代码会遍历所有之前收集的线程，调用cgroup_task_migrate进行相关数据结构的调整工作，第6行～第10行代码遍历当前cgroupfs_root下所有cgroup_subsys实例的attach回调方法进行子系统特定的加入逻辑。

最后，我们看下cgroup_task_migrate函数的具体逻辑：

```
1. static void cgroup_task_migrate(struct cgroup *old_cgrp,
2.                                 struct task_struct *tsk,
3.                                 struct css_set *new_cset)
4. {
5.     struct css_set *old_cset;
6.
7.     WARN_ON_ONCE(tsk->flags & PF_EXITING);
8.     old_cset = task_css_set(tsk);
9.
10.    task_lock(tsk);
11.    rcu_assign_pointer(tsk->cgroups, new_cset);
12.    task_unlock(tsk);
13.
14.    write_lock(&css_set_lock);
15.    if (!list_empty(&tsk->cg_list))
16.        list_move(&tsk->cg_list, &new_cset->tasks);
17.    write_unlock(&css_set_lock);
18.
19.    set_bit(CGRP_RELEASABLE, &old_cgrp->flags);
20.    put_css_set(old_cset);
21.}
```

第10行～第12行代码将线程tsk的cgroups成员指向新的css_set数据结构实例new_cset。

第14行～第17行代码反向绑定新css_set实例new_cset与tsk的关联，即将tsk加入到new_cset的tasks链表中。

第19行～第21行代码进行老的css_set实例的释放相关工作。

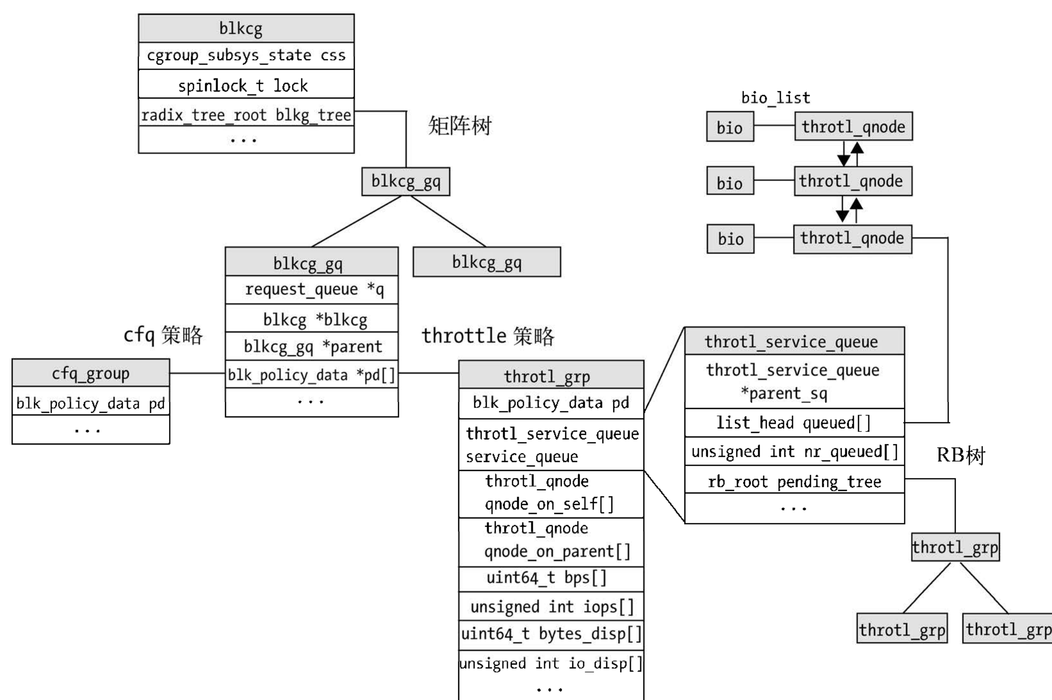
A.3.4　Cgroup block子系统IO限流实现

前面我们分析了CGroup框架相关的实现，还有很多细节逻辑，有兴趣的读者可以自行翻阅源代码学习。下面我们以CGroup block子系统下的限流模块throttle为例，介绍下实现一个CGroup

子系统需要完成哪些事情。

整个block子系统提供的资源控制包括两部分，一部分是限流功能，另一部分是带宽分配功能。限流功能的代码位于block/blk-throttle.c文件中，带宽分配功能的代码位于block/cfq-iosched.c文件中。另外，block/blk-cgroup.c及block/blk-cgroup.h文件中包含一部分通用逻辑。

block限流子系统的功能在本章开始的时候已经介绍过，这里我们可以为不同的设备分配不同的bps（每秒读写流量）或者iops（每秒读写次数），CGroup系统会限制相应设备的流量以满足用户设置的要求。要理解block子系统的限流功能，首先需要了解整个block子系统在CGroup方面所做的工作。整个block子系统的CGroup实现的核心数据结构关系如图A-16所示。



图A-16　block子系统的CGroup核心数据结构

整体block子系统的CGroup核心数据结构的关系比较复杂，很难在一张图中完全画清楚，图A-16仅仅展示了其中的一部分。下面我们详细分析涉及的所有核心数据结构，其中一部分是图A-16中没有展示出来的。

首先，我们知道所有实现CGroup的各个功能的子系统都会有一个`cgroup_subsys_state`的实例，那对于block子系统来说，这个最核心的数据结构是`blkcg`。下面我们给出该数据结构的定义：

```
struct blkcg {
    struct cgroup_subsys_state    css;
    spinlock_t                    lock;

    struct radix_tree_root        blkg_tree;
    struct blkcg_gq          *blkg_hint;
    struct hlist_head        blkg_list;

    uint64_t            id;

    unsigned int        cfq_weight;
    unsigned int        cfq_leaf_weight;
};
```

下面简要介绍该数据结构中各个成员的含义。

- 该数据结构的首个成员css是cgroup_subsys_state类型。
- lock用于帮助实现临界区的保护。
- blkg_tree是一颗基树，其中以block子系统的核心数据结构request_queue的id为键，存放的是blkcg_gq这个数据结构的实例。这里简单解释下，因为block子系统的CGroup大多是配置在某个块设备上，在同一个资源组内我们通常可以为不同的设备分配不同的策略，并且在block层的CGroup策略并不只有一种，主要有cfq权重控制及throttle限流控制两种方式，它们的实现也完全不同。所以，这里的blkg_tree存放的是该资源组下涉及的所有设备的一些策略配置信息，这个策略配置信息的数据结构就是blkcg_gq。后面我们会详细分析这个数据结构。
- blkg_hint是一个用于快速查找的缓存。每次从blkg_tree基树中查找成功时，根据需要可能会将查找结果存放到此成员上，下次查找直接可以从这个成员上获取。
- blkg_list与blkg_tree的功能类似，区别在于后者将blkcg_gq以基树的形式组织，后者将blkcg_gq以链表的形式组织，方便一些遍历操作。
- id是用于唯一地标识该blkcg实例。
- cfq_weight及cfq_leaf_weight这两个成员与cfq相关，我们这里不做讨论。

下面我们再来看一下blkcg_gq这个数据结构：

```
struct blkcg_gq {
    struct request_queue         *q;

    struct list_head         q_node;
    struct hlist_node        blkcg_node;

    struct blkcg             *blkcg;
    struct blkcg_gq          *parent;

    struct request_list      rl;
```

```
    int                        refcnt;
    bool                     online;

    struct blkg_policy_data  *pd[BLKCG_MAX_POLS];

    struct rcu_head          rcu_head;
};
```

下面简要介绍一下这个数据结构中的成员。

- q是一个request_queue类型的指针，指向相应设备的请求队列。
- q_node用于实现链表。
- blkcg_node是一个单链表域。前面讲过的blkcg数据结构中的blkg_list链表头保存了该资源组中所有的blkcg_gq实例，blkcg_node域用于实现该链表。
- blkcg用于指向blkcg_gq所属的blkcg实例。
- parent指向父blkcg_gq实例。
- rl成员用于分配request实例。
- refcnt是该数据结构的引用计数器。
- online标志标识了该实例是否已经是在线状态。
- pd指针数组是该数据结构中非常重要的一个成员，它的作用类似于前面讲过的CGroup通用框架里的cgroup_subsys_state类型，代表了一个抽象数据结构，用于帮助内核代码完成一个多态的数据结构。对于通用块层的资源控制，通常的策略是按照每资源组每设备进行的，并且可能同时存在多种策略，例如权重策略与限流策略是完全不同的，所以这里抽象了一个相当于面向对象设计思想里面的基类数组，不同策略定义自己的核心数据结构，并存入该数组中。
- rcu_head这个成员在限流子系统中基本没有用到，这里可以先忽略。

下面我们马上看下blkg_policy_data数据结构的定义：

```
struct blkg_policy_data {
    struct blkcg_gq       *blkg;
    int                   plid;

    struct list_head      alloc_node;
};
```

下面简要介绍该数据结构中各成员的定义。

- blkg指向blkcg_gq数据结构的实例。
- plid的含义是policy id，即策略ID，每个策略子系统都有自己固定的ID。该成员初始化好后，就不再改变。
- alloc_node是一个链表域，在部分代码中用于链接相关的blkg_policy_data实例。

通用块层的CGroup策略实现主要有权重和限流这两个策略。权重策略依赖于CFQ调度器，限流是单独的一个模块。权重策略的blkg_policy_data实现的是cfq_group数据结构，限流策略的blkg_policy_data实现是throtl_grp数据结构。这两个数据结构的定义中首个成员都是blkg_policy_data实例，以此来完成多态设计。在具体了解策略实现之前，我们还要看一个重要的数据结构——blkcg_policy，该数据结构用于不同策略的具体实现：

```
struct blkcg_policy {
    int                     plid;
    size_t                  pd_size;
    struct cftype           *cftypes;

    blkcg_pol_init_pd_fn            *pd_init_fn;
    blkcg_pol_online_pd_fn   *pd_online_fn;
    blkcg_pol_offline_pd_fn  *pd_offline_fn;
    blkcg_pol_exit_pd_fn            *pd_exit_fn;
    blkcg_pol_reset_pd_stats_fn      *pd_reset_stats_fn;
};
```

下面简要介绍该数据结构中各个成员的含义。

- plid与blkcg_policy_data中plid的含义相同，代表一个固定的策略编号。
- pd_size代表该策略的核心数据结构的大小。因为我们通过blkg_policy_data来表示具体策略的基类，但是具体策略的实现不同，其核心数据结构的大小也会不相同，所以在这里保存了核心数据结构的大小，用于分配具体策略子系统的核心数据结构。
- cftypes是cftype类型的，这在之前已分析过。这个成员存放了各个策略子系统对应的CGroup配置文件。
- pd_init_fn用于初始化具体策略子系统。
- pd_online_fn在子系统投入工作时会被调用。
- pd_offline_fn在子系统被卸载时使用。
- pd_exit_fn在子系统退出时调用。
- pd_reset_stats_fn在子系统被重置时调用，对应CGroup的reset_stats文件。

下面我们开始分析block子系统限流模块涉及的一些核心数据结构，首先我们给出了throtl_grp这个数据结构的定义：

```
struct throtl_grp {
    struct blkg_policy_data pd;

    struct rb_node rb_node;

    struct throtl_data *td;

    struct throtl_service_queue service_queue;
```

```
    struct throtl_qnode qnode_on_self[2];
    struct throtl_qnode qnode_on_parent[2];

    unsigned long disptime;

    unsigned int flags;

    bool has_rules[2];

    uint64_t bps[2];

    unsigned int iops[2];

    uint64_t bytes_disp[2];

    unsigned int io_disp[2];

    unsigned long slice_start[2];
    unsigned long slice_end[2];

    struct tg_stats_cpu __percpu *stats_cpu;

    struct list_head stats_alloc_node;
};
```

下面简要介绍一下这个数据结构中各个成员的含义。

- pd我们已经知道，它帮助我们完成不同策略的多态实现。
- rb_node是一个红黑树的链接域，用于将throtl_grp链接到后面要分析的throtl_service_queue的pending_tree中，以具体bio（块读写）的到期时间作为键，即按throtl_grp里存放的bio请求的到期时间来排序。
- td存放限流子模块中一些特定的数据结构，后面会详细分析。
- service_queue成员存放了该资源组中所有被限流的bio请求，同时包含了一颗红黑树用于帮助完成bio请求的排序工作。后面我们会详细分析这个数据结构。
- qnode_on_self及qnode_on_parent是throtl_qnode类型的数组，它们都有两项，分别代表读和写，数组主要存放被限流的bio请求。由于CGroup框架本身是支持多层级树形结构的，这种情况下位于最下层的叶子节点的资源组的bio请求被放行，需要继续派发到上一层的资源组中，看是否超出父资源组的IO限流要求，我们需要区分来源于自己资源组本身的IO请求与来源于下一层派发上来的IO请求，避免由于来自下一层派发的请求过多导致本资源组自身的IO请求得不到满足，以致出现严重的不公平现象，所以通过引入两个单独的成员，qnode_on_self存放自身资源组中的IO请求，qnode_on_parent存放子资源组派发上来的IO请求，这样就可以有效避免不公平现象的发生。
- disptime代表的是该资源组下次可以派发IO的时间，单位是 jiffies。

- flags记录该资源组的一些状态标志，可能的值是THROTL_TG_PENDING和THROTL_TG_WAS_EMPTY，它们分别代表该资源组处于待处理状态或者是该资源组没有被限流等待派发的IO请求。
- has_rules数组代表的是该资源组分别在读和写上有没有限流规则存在，用于在提交IO请求后，判断是否要对该IO进行限流等。
- bps数组代表的是该资源组在读和写上每秒钟允许的字节数，iops数组代表的是读和写上每秒钟允许的执行数量。
- bytes_disp数组代表的是当前资源组在这一秒内已经流过的IO字节数，io_disp数组代表的是当前资源组在这一秒内已经通过的读写次数。
- slice_start和slice_end数组用于内部时间相关的计算工作。
- stats_cpu用于存放每个CPU各自的一些统计数据，例如总共传输了多少字节的数据，发生了多少次读写请求等。
- stats_alloc_node用于分配stats_cpu数据结构。

下面我们再详细看看throtl_grp数据结构中涉及的一些其他数据结构的定义，首先来了解一下throtl_service_queue这个数据结构：

```
struct throtl_service_queue {
    struct throtl_service_queue *parent_sq;
    struct list_head   queued[2];
    unsigned int nr_queued[2];

    struct rb_root  pending_tree;
    struct rb_node  *first_pending;
    unsigned int nr_pending;
    unsigned long first_pending_disptime;
    struct timer_list  pending_timer;
};
```

下面我们简要介绍该数据结构中各个成员的含义。

- parent_sq指向父throtl_service_queue数据结构的实例。
- queued链表数组用于存放在读和写上的bio请求，链表的内容是throtl_qnode数据结构的实例。
- nr_queued数组用于统计在queued链表数组中读和写的项数。
- pending_tree是一颗红黑树的根节点，将throtl_grp数据结构实例组织成一颗红黑树，以throtl_grp的disptime进行排序。
- first_pending与first_pending_disptime分别存放了红黑树中首个等待派发的throtl_grp及其具体的派发时间disptime，主要用于快速查找。
- nr_pending存放的是等待派发的资源组throtl_grp的个数。
- pending_timer成员是一个定时器，用于在指定时间触发相关函数执行。实际上，这里是throtl_pending_timer_fn函数。

下面再看一下throtl_qnode数据结构：

```
struct throtl_qnode {
    struct list_head   node;
    struct bio_list       bios;
    struct throtl_grp     *tg;
};
```

下面简要介绍该数据结构中各个成员的含义。

- node是一个链表的链接域，用于将throtl_qnode实例链接在service_queue的queue数组中。
- bios是相关bio请求的实例。
- tg指向所属的资源组throtl_grp的实例。

限流子模块涉及的核心数据结构基本就是这些，下面我们开始分析该子模块的功能实现逻辑，首先以限流子模块的初始化过程开始分析。

限流子模块的初始化涉及内核代码流程的两个部分。第一部分为模块初始化阶段，它通过throtl_init函数进行限流子模块的一些数据结构分配及策略注册等工作。第二部分为块设备驱动注册的过程中，当为块设备分配相应的请求队列request_queue时，会调用限流子模块针对设备请求队列单独的初始化方法blk_throtl_init。第一部分只在模块初始化阶段调用一次，第二部分在每个块设备分配请求队列时都会被调用。下面我们首先分析throtl_init函数的逻辑：

```
1. static int __init throtl_init(void)
2. {
3.      kthrotld_workqueue = alloc_workqueue("kthrotld", WQ_MEM_RECLAIM, 0);
4.      if (!kthrotld_workqueue)
5.          panic("Failed to create kthrotld\n");
6.
7.      return blkcg_policy_register(&blkcg_policy_throtl);
8. }
```

第3行代码分配一个名字为kthrotld的工作队列，该工作队列用于限流子模块异步派发bio等工作，后面会详细分析。

第7行代码用于注册限流子模块策略。在熟悉blkcg_policy_register的注册逻辑前，我们先来看下blkcg_policy_throtl的定义：

```
// blk-throttle.c

static struct blkcg_policy blkcg_policy_throtl = {
    .pd_size         = sizeof(struct throtl_grp),
    .cftypes         = throtl_files,

    .pd_init_fn      = throtl_pd_init,
    .pd_online_fn    = throtl_pd_online,
```

```
    .pd_exit_fn       = throtl_pd_exit,
    .pd_reset_stats_fn  = throtl_pd_reset_stats,
};
```

下面简要介绍该数据结构中各个成员的含义。

- blkcg_policy数据结构前面我们分析过，通过代码可以看到限流子模块核心数据结构的大小pd_size就是throtl_grp的大小。
- cftypes域存放了限流子模块独立的配置文件，这里不再列出。
- throtl_pd_init、throtl_pd_pnline、throtl_pd_exit、throtl_pd_reset_stats等相关回调函数在限流子模块的不同时机会被调用，后面我们会详细分析。

下面我们再看下blkcg_policy_register函数的逻辑：

```
// blk-cgroup.c

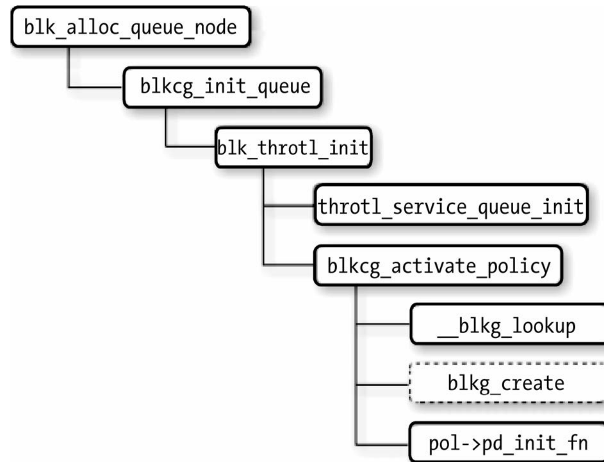
1. int blkcg_policy_register(struct blkcg_policy *pol)
2. {
3.     int i, ret;
4.
5.     if (WARN_ON(pol->pd_size < sizeof(struct blkg_policy_data)))
6.         return -EINVAL;
7.
8.     mutex_lock(&blkcg_pol_mutex);
9.
10.    ret = -ENOSPC;
11.    for (i = 0; i < BLKCG_MAX_POLS; i++)
12.        if (!blkcg_policy[i])
13.            break;
14.    if (i >= BLKCG_MAX_POLS)
15.        goto out_unlock;
16.
17.    pol->plid = i;
18.    blkcg_policy[i] = pol;
19.
20.    if (pol->cftypes)
21.        WARN_ON(cgroup_add_cftypes(&blkio_subsys, pol->cftypes));
22.    ret = 0;
23.out_unlock:
24.    mutex_unlock(&blkcg_pol_mutex);
25.    return ret;
26.}
```

第5～6行代码判断待注册的策略的核心数据结构的大小是否正常。前面讲过，通用块层CGroup的不同策略子系统通过将其核心数据结构的首个成员统一为blkg_policy_data类型来实现多态，所以这里基本的判断就是这个数据结构至少不能小于blkg_policy_data的大小。

第11 ~ 18行代码从系统全局数组blkcg_policy中查找一个空闲的槽位来存放待注册策略的数据结构实例。如果没有空闲槽位，则跳到out_unlock处，返回ENOSPC错误。如果找到空闲槽位，则将策略数据结构实例放入相应槽位中。

第20 ~ 21行代码通过cgroup_add_cftypes函数将限流策略独立的配置文件信息加入到系统中。

第一部分的初始化主要完成策略注册的相关工作。下面我们来看每个设备加入系统后，限流子模块针对每个设备的初始化逻辑，这部分逻辑代码在blk_throtl_init函数中。我们知道，每个块设备在注册自己的驱动程序时，都需要创建和初始化块设备相关的核心数据结构request_queue，那么限流子模块针对每个设备的初始化工作就是在设备驱动分配自己的request_queue时进行的。整体的调用关系如图A-17所示。



图A-17 blk_alloc_queue_node调用关系

blk_alloc_queue_node函数用于块设备驱动分配自己的请求队列，中间的blkcg_init_queue是一个简单的封装函数，只是简单调用blk_throtl_init函数。在blk_throtl_init函数之前所做的工作属于块设备驱动范畴，这里我们不做分析，从blk_throtl_init函数开始分析：

```
1. int blk_throtl_init(struct request_queue *q)
2. {
3.     struct throtl_data *td;
4.     int ret;
5.
6.     td = kzalloc_node(sizeof(*td), GFP_KERNEL, q->node);
7.     if (!td)
8.         return -ENOMEM;
9.
10.    INIT_WORK(&td->dispatch_work, blk_throtl_dispatch_work_fn);
11.    throtl_service_queue_init(&td->service_queue, NULL);
```

```
12.
13.    q->td = td;
14.    td->queue = q;
15.
16.    ret = blkcg_activate_policy(q, &blkcg_policy_throtl);
17.    if (ret)
18.        kfree(td);
19.    return ret;
20.}
```

第3~8行代码用于分配throtl_data数据结构实例，第10行代码用于初始化限流IO派发函数为blk_throtl_dispatch_work_fn。

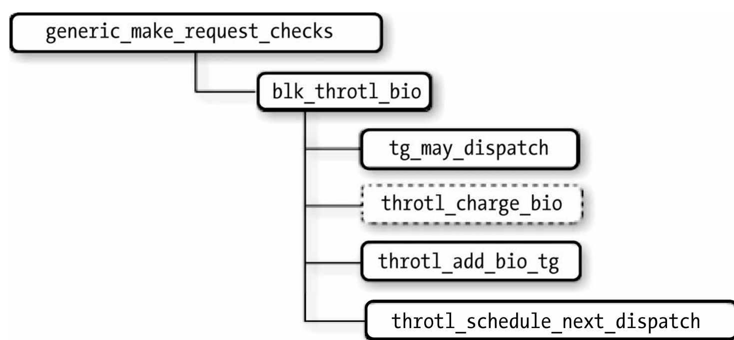
第11行代码用于初始化限流子模块的service_queue，其中除了初始化各个成员外，还要初始化一个定时器，用于指定IO限流时间到后的定时触发工作。

第16行代码通过blkcg_activate_policy进行策略的激活操作。

在函数blkcg_activate_policy中，进行各层blkcg_gq数据结构的分配和建立工作，主要通过调用__blkg_lookup与blkg_create函数来完成。

下面我们简要介绍一下限流功能执行的主要代码路径。

一个具体的IO请求从VFS层提交给通用块层的接口函数是submit_bio。在进行了简单的判断和统计工作后，会调用generic_make_request函数，这个函数内部开始根据请求bio对应的设备来调用相应独立的make_request_fn函数，在此之前有一些简单的判断工作，限流子模块的功能就是在这里实现的。具体对应的函数是generic_make_request_checks（其调用流程如图A-18所示）。



图A-18 generic_make_request_checks调用流程

限流功能的大体逻辑如下所示。

(1) 通过tg_may_dispatch函数判断当前时间范围内统计的bps和iops是否超过限额，如果没有则调用throtl_charge_bio工作处理一些时间片以及统计当前流过的bps和iops。

(2) 如果tg_may_dispatch函数判断出已经超过单位时间内的限额，则将当前bio请求加到相应的待处理队列中，等待指定时间到达后触发定时器事件，将被限流的bio派发出去，定期时间到达后，具体触发的函数是throtl_pending_timer_fn。每次进入generil_make_request_checks函数或者当等待派发的队列为空时，会通过throtl_schedule_next_dispatch函数来启动或更新定时器事件的触发时间。

由于篇幅所限，这里省略了很多具体限流逻辑的一些细节处理，感兴趣的读者可以自行翻阅相关代码。

欢迎加入

图灵社区 iTuring.cn

——最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

优惠提示：现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。

——最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

——最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

MariaDB
原理与实现

本书对于MariaDB的各种特性如数家珍，从基本原理到源代码，使读者洞悉一切。但这又不是一本只讲MariaDB的书，还详细介绍了MariaDB与MySQL的差异，而且作者还无私分享了很多京东数据库的实践经验。总之一句话，这是一本关于MariaDB与众不同的好书！

——汪源，网易杭州研究院副院长

之前国内在数据库系统方面的技术积累有限，现在我看到国内越来越多数据库方面图书的出版，感到由衷的高兴。

作为MySQL的一个重要源代码分支，MariaDB包括的一些新特性使它在某些地方优于MySQL，同时基于对MySQL后续发展的担心，包括谷歌在内的一些大公司开始逐步使用MariaDB替换MySQL。

本书从设计原理开始讲解，深入浅出地剖析了MariaDB以及MySQL的几部分重要功能的具体实现细节，是作者在京东工作实践中的提炼和总结。通过本书，你不但可以学习到数据库方面的基础理论，而且也可以参照书中的实例进行MariaDB源代码方面的实践。相信本书对使用和学习MariaDB的读者有很大的帮助。

——杨海朝，MySQL ACE Director/新浪首席数据库架构师

MariaDB不仅免费开源，而且支持一些新特性，如多源复制、线程池、binlog group commit等。另外，Google已经从MySQL迁移到了MariaDB。

本书由浅入深地讲述了MariaDB的基本原理和实现细节，并结合在京东的实战经验，给出基于MariaDB的分布式数据库的最佳实现。

相信本书对学习MariaDB的读者会非常有帮助。

——吴元清，京东架构师

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/数据库/MariaDB
人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-38517-8
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks